NOTE TO USERS

The original manuscript received by UMI contains pages with indistinct and/or slanted print. Pages were microfilmed as received.

This reproduction is the best copy available

UMI

University of Alberta

Visualizing Object/Method Granularity for Program Parallelization

by

William L.M. Hui ©

A thesis submitted to the Faculty of Graduate Studies and Research in partial fulfillment of the requirements for the degree of **Master of Science**.

Department of Computing Science

Edmonton, Alberta
Spring 1998

National Library of Canada
Acquisitions and Bibliographic Services
395 Wellington Street
Ottawa ON K1A 0N4
Canada

Bibliothèque nationale du Canada
Acquisitions et services bibliographiques
395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence

Our file Notre référence

The author has granted a non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of this thesis in microform, paper or electronic formats.

The author retains ownership of the copyright in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

L'auteur a accordé une licence non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de cette thèse sous la forme de microfiche/film, de reproduction sur papier ou sur format électronique.

L'auteur conserve la propriété du droit d'auteur qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

0-612-28948-6

Canada

University of Alberta

Faculty of Graduate Studies and Research

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies and Research for acceptance, a thesis entitled **Visualizing Object/Method Granularity for Program Parallelization** submitted by William L.M. Hui in partial fulfillment of the requirements for the degree of **Master of Science**.

. . . . . .
Duane Szafron

. . . . . .
Jonathan Schaeffer

. . . . . .
Ben Watson

. . . . . .
Dennis Ward

Date: Jan 19, 98

To my parents;

they brought me, a dozen years ago, an Apple][+
with which I wrote my first computer program.

Abstract

Demands for more computing power are increasing. Parallel hardware, which provides a large amount of computing power, is now commonplace and is a cost-effective solution. However, this type of hardware is not being fully utilized since parallel programming is not a user-friendly task to many developers. To make matters worse, parallel programming becomes more complex when it comes to the object-oriented (OO) paradigm. We believe that one of the solutions is to build tools that help programmers convert their sequential OO programs to parallel ones.

Revy is a platform-neutral, easy-to-use, object/method granularity visualization system that helps inexperienced parallel programmers transform their sequential OO programs into parallel ones. *Revy* is a program visualization system that allows users to view and inspect the object communication patterns of their applications. *Revy* is also a profiling system which helps users identify the high-granularity objects/methods as candidates for remote execution. *Revy* achieves these objectives by providing an annotation subsystem which instruments any user's classes and methods. The user's application is then compiled and executed to produce the granularity information as well as the object communication patterns, which are then processed and visualized using a graphical user interface. In this thesis, we present the requirements, architecture, design and implementation of *Revy*. We also discuss some problems and concerns, and suggest a few further enhancements. Finally, the experimental results that confirm the usefulness of *Revy* are presented.

Acknowledgements

First, I would like to thank Professor Szafron and Professor Schaeffer for their creative ideas, knowledgeable feedback, strict writing rules, and the fun and jokes since I have joined their team. I appreciate their time and effort, and I want to thank them for being patient to me. Thanks to Professor Unrau, for his harsh but valid comments, from which I have learnt a lot. Next, to the the members in the Parallel Lab. To Steve MacDonald, thanks for his comments, and we had quite a few good discussions on Java. To Diego Novillo and Mark Brockington, thanks for all the amusement. Also, I thank Carol Smith, John Bartoszewski, Steve Sutphen, and the Operations Group. I am impressed by their responsiveness. They helped me a lot in my research, especially in the experimental part.

I would also like to thank my friends Vivien Chu, Alan Kwok, Andrew Leung and Benjamin Ngo, who patiently proof-read my papers. I am also very grateful to the Ching family for keeping me in good health and caring about my progress. In fact, they have always been treating me like a dear member in their family.

Most of all, I am indebted to Juliana Ching, my fiancée, for being patient, helpful and caring to me during the course of my study. I must thank God for giving me such a knowledgeable mind and prayful heart such that we can, as the Bible says, "rejoice with those who rejoice, and weep with those who weep" (Romans 12:15).

Lastly and the most importantly, I want to thank my lord Jesus. *Soli Deo Gloria* – "To God alone be the Glory".

Contents

1 Introduction **1**

1.1 The Revy System 3

1.2 Obtaining the Revy Distribution 5

1.3 Thesis Organization 5

2 Related Works **6**

2.1 Visualization 6

2.1.1 Enterprise 6

2.1.2 Eiffel // 7

2.2 Granularity 9

2.2.1 Profiling Tools 9

2.2.2 Critique 10

2.3 Program Parallelization 11

2.3.1 Parallelization Algorithms 11

2.3.2 Parallelizing Compilers 12

2.4 Summary 13

3 Revy System **14**

3.1 Requirements 14

3.2 Architecture 15

3.3 Architecture Overview 15

3.4 Architecture Details 17

3.5 Design 18

3.5.1 The Data Flow Design 18

3.5.2 The State Design 19

3.6 Implementation 20
3.7 Summary 20

4 Visualization and Interaction Subsystem **21**
4.1 Requirements 21
4.2 Architecture 22
4.3 Design 23
4.4 Revy Graphical User Interface 24
4.4.1 Revy Main Window 24
4.4.2 Defining a New Project 25
4.4.3 Annotating User's Code 26
4.4.4 Compiling and Executing 28
4.4.5 Visualizing the Runtime Object Call Graph 29
4.4.6 Animating Execution 31
4.4.7 Filtering Instances and Messages 32
4.4.8 Highlighting Instances and Messages 32
4.4.9 Displaying Runtime Statistics 34
4.5 Further Enhancements 35
4.6 Summary 36

5 Parsing and Annotation Subsystem **37**
5.1 Requirements 37
5.2 Architecture 39
5.2.1 PAS Architecture 40
5.2.2 Architectural Strategies 41
5.3 Design 42
5.3.1 Interface Files 42
5.3.2 Class, Responsibility and Collaborators 43
5.3.3 Collaboration Diagrams 43
5.4 Implementation 45
5.4.1 Type Table 45
5.4.2 Annotating Classes 47

5.4.3 Object Size 47
5.4.4 Annotating Methods 48
5.4.5 Message Size 50
5.5 Problems and Concerns 51
5.5.1 Problems with Tools 51
5.5.2 Language Problems 53
5.5.3 Probe Effects 55
5.6 Further Enhancements 56
5.7 Summary 57

6 Runtime Modeling Subsystem **59**
6.1 Requirements 59
6.2 Architecture 60
6.3 Design 61
6.3.1 Dynamic Trace File 61
6.3.2 Class, Responsibility and Collaborators 62
6.3.3 Collaboration Diagrams 62
6.4 Implementation 64
6.4.1 Logging Runtime Traces 64
6.4.2 Translating Traces 64
6.4.3 Call Graph and Statistics 66
6.5 Concerns and Further Enhancements 66
6.5.1 `RevyLogger` 66
6.5.2 `DTFReader` 67
6.5.3 `ObjectGraph` 68
6.6 Summary 69

7 Experimental Results **70**
7.1 Test Bed 70
7.2 DTFReader 71
7.2.1 Program Description 71
7.2.2 Runtime Heuristics 71

7.2.3 Parallelization Results 72
7.2.4 Probe Effects 74
7.3 HTML File Parser 76
7.3.1 Program Description 76
7.3.2 Runtime Heuristics 76
7.3.3 Parallelization Results 77
7.3.4 Probe Effects 79
7.4 A Zip Application 81
7.4.1 Program Description 81
7.4.2 Runtime Heuristics 81
7.4.3 Parallelization Results 84
7.4.4 Probe Effects 85
7.5 An Object Adder 86
7.5.1 Program Description 86
7.5.2 Performance Analysis 88
7.6 Guidelines for Parallelization 90
7.7 Summary 93

8 Conclusion **94**
8.1 Future Work 94
8.1.1 Visualization and Interaction Subsystem 94
8.1.2 Parsing and Annotation Subsystem 94
8.1.3 Runtime Modeling Subsystem 95
8.2 Summary 95

List of Figures

1.1 Granularity of a Method 3

3.1 System Architecture in Brief 16
3.2 System Architecture in Detail 17
3.3 Data Flow Diagram of Revy 19
3.4 State Diagram of Revy 20

4.1 VIS Architecture 22
4.2 Revy Window 25
4.3 Button Panel 26
4.4 Project Dialog Window 26
4.5 Annotation Dialog Window 27
4.6 Compile Dialog Window 28
4.7 Execute Dialog Window 28
4.8 Visualizing an Object Call Graph 30
4.9 Animating Execution 31
4.10 Animation Dialog Window 31
4.11 Filtering Dialog Window 33
4.12 Highlight Dialog Window 33
4.13 Displaying Runtime Statistics 34

5.1 Method Annotation 38
5.2 Trace Output 39
5.3 PAS Architecture 40
5.4 Parsing And Extracting Static Artifacts 44
5.5 Annotating a User's Method 45

5.6 An Example of Finding Static Object Size 49
5.7 Finding Method Size by Sampling 51
5.8 Finding the Exact Method Size 52
5.9 Catching All Exceptions 57
5.10 Method Signatures 58

6.1 RMS Architecture 60
6.2 Format of the Dynamic Trace File 61
6.3 Logging Runtime Traces 62
6.4 Translating Traces 63
6.5 Querying Statistics 63
6.6 Identifying Message Sender 65

7.1 Object Graph of the DTFReader Application 73
7.2 Runtime Statistics of the DTFReader Application 73
7.3 Plottings of the DTFReader Probe Effects 75
7.4 Object Graph of the HTMLParser Application 78
7.5 Runtime Statistics of the HTMLParser Application 78
7.6 Plottings of the HTMLParser Probe Effects 80
7.7 Object Graph of the JHLZip Application 82
7.8 Runtime Statistics of the JHLZip Application 82
7.9 Parallelizing `ZipFileHeader` 83
7.10 Net and Average Active Times of the Messages of JHLZip 83
7.11 Plottings of the JHLZip Probe Effects 86
7.12 The ObjectAdder Program 87

List of Tables

7.1 Performance of the Parallel DTFReader 74
7.2 Timings of the Probe Effects of DTFReader 75
7.3 Performance of the Parallel HTMLParser 79
7.4 Timings of the Probe Effects of HTMLParser 80
7.5 Performance of the Parallel JHLZip 84
7.6 Performance of the JHLZip Parallelized with MethodThread 85
7.7 Timings of the Probe Effects of JHLZip 86
7.8 Performance of the ObjectAdder with One Thread 90
7.9 Performance of the ObjectAdder with Two Threads 90
7.10 Performance of the ObjectAdder with Four Threads 91

Chapter 1

Introduction

Demands for more computing power are increasing. Networked and multiprocessor workstations provide users with a large amount of computing power by executing the user's application on many processors at once. This type of hardware is now commonplace and is a cost-effective approach to solving computationally-intensive applications. However, the software advances necessary to exploit this power have lagged behind the hardware advances.

Writing parallel software is often perceived as a complicated endeavor. The design, implementation and testing of parallel software is more difficult than comparable sequential software. Parallel programming extends the complexity of sequential programming to include issues such as synchronization, deadlock, non-deterministic program behaviour due to concurrent execution, and the communication and granularity issues in the remote execution of tasks.

In the Object-Oriented (OO) programming paradigm, writing parallel programs encounters additional problems than in traditional procedural languages. OO systems model real world objects, which are generally autonomous entities whose activities are performed concurrently. Thus, objects and parallel programs seem to be a perfect match. However, this is not always the case. For instance, the problems with inheritance, encapsulation, and reusability are identified as the outstanding research issues for concurrent OO language designers.

Several researchers [KL89, MY94] have pointed out that the main problem in concurrent object programming is the conflict of the inheritance and synchronization constraints. These two constraints often interfere with each other, resulting in the *Inheritance Anomaly*, where the re-definition of inherited methods is necessary in order to maintain the integrity of concurrent objects. Parallel object-based languages therefore either do not support inheritance [AM87], or do so only by compromising the encapsulation [YT86] or the reusability [CA88] properties.

Observing that there are difficulties in concurrent OO programming, we need tools today which help parallel software developers build their OO systems easily and efficiently. The dream tool for concurrent OO programming would be a parallelizing compiler that automatically transforms a sequential program to a parallel version, for example, "`g++ -parallel myFile.cc`". However, such a tool is still many decades away. This thesis proposes an alternative approach, a parallelization advisor, that assists programmers in parallelizing OO applications.

Revy is an integrated system that provides inexperienced parallel programmers with an easy-to-use tool to help them convert their sequential OO programs into parallel ones. [WW94] pointed out that there is a need for a tool that allows one to visualize the object communication patterns to help in the development of concurrent OO programs. Revy is such a program visualization system. Revy is also a profiling system. With the help of Revy's profiling feature, users can easily identify the objects/methods to be parallelized, by analyzing the *granularity* information provided.

Chandy and Taylor [CT92] state that "the granularity of a computation is the ratio of computation to communication". This is illustrated in Figure 1.1. Executing the method `foo()` remotely on another computer frees up CPU cycles on the local machine, but results in communication overheads. The overheads include the time spent on directing the remote processor to invoke the method, as well as packing, sending, receiving, and unpacking the data. Therefore, programmers must weigh the benefits of freeing up the local processor versus the costs of the communication overhead.

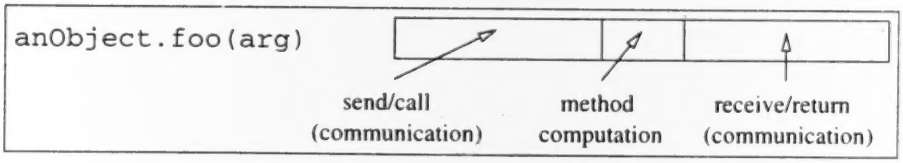


Figure 1.1: Granularity of a Method

In object-oriented systems, granularity plays an even more important role since OO programs are typically fine-grain in nature. In other words, OO programs often have a low computation to communication ratio. Developers tend to implement many short methods instead of a few long ones to increase the reusability and maintainability of their software. To enhance encapsulation, they introduce many simple classes. For example, a phone number is sometimes declared as a class rather than simply a string of 10 digits. As a result, one can see that managing granularity is critical to efficient OO programming.

1.1 The Revy System

As an integrated parallelization advisor, Revy is built with three research objectives in mind:

1. to explore visualization techniques that help users understand the object communication patterns of their OO programs,

2. to implement profiling techniques for collecting and processing the granularity and trace information of an OO program, so that higher-grain objects/methods can be identified for parallelization, and

3. to investigate some parallelization heuristics which may be implemented to (more) correctly suggest the candidate objects/methods for parallelization.

Revy has two features that distinguish it from traditional profiling and program visualization tools. First, Revy considers not only the timing statistics, but also the object and method parameter sizes of a user program. Such information is essential

because applications distributed over a network of workstations always involve communication overheads which mainly consist of the time spent on transmitting data. Second, the granularity information collected and displayed is instance-based, as opposed to function-based data featured by traditional profilers. Such instance-based data is important for programmers to better understand the behaviour of their OO programs. Understanding the runtime behaviour plays an important role in efficiently parallelizing an OO program.

Revy takes a user program as input, parses it to identify the classes and methods, instruments it so that it can gather statistical and trace information at runtime, compiles it and executes it to obtain the runtime statistics. From the trace information, Revy builds an object call graph, which is a runtime interaction diagram. Revy then processes the runtime statistics which, together with the object call graph, are visually reported to the user. The user program can be either a sequential or a parallel program with threads. Revy has a generic OO programming model that analyzes C++ or Java programs.

Since Revy employs software instrumentation to measure the granularity, it inevitably suffers from a phenomenon called the *probe effect*. [GW95] defines this as being "the way in which collecting information about the execution of the system may alter that system's behaviour". For example, unless special hardware exists to perform the measurement, every processor will have to spend a few extra cycles logging this information during each operation. This can easily change the timing of certain events, and alter the program's overall behaviour.

Revy can be used for numerous purposes. First, Revy aids OO programmers to parallelize their system by visualizing the granularity information. The information collected is instance-based and includes: object and message processing times, object and message parameter sizes, and so on. Using visual information such as an object call graph and granularity data, developers can easily apply some parallelization heuristics (such as graph partitioning algorithms).

Second, Revy can be used as a profiling tool. Its feature of collecting and visualizing a program's runtime statistics serves well for performance tuning. Revy provides timing statistics, such as the processing times of objects and messages, so developers can identify the performance bottlenecks of their application. As a result, Revy points the users to the places in their code where performance tuning is needed whether their programs are to be parallelized or not.

Third, Revy can be used as a reverse engineering tool that improves program understanding for software maintenance. Revy reverse-engineers a program from its source code to a runtime object interaction diagram. As a result, users can have a better understanding of the design of the object relationships and communication patterns of their code.

1.2 Obtaining the Revy Distribution

The Revy system can be downloaded via the World Wide Web. The distribution includes this thesis, the Java source and class files, a few batch files, and some example programs. Chapter 4 of the thesis is the user manual for the system. The download site is: `http://www.cs.ualberta.ca/~whui/revy/`.

1.3 Thesis Organization

This thesis consists of eight chapters. Chapter 2 compares Revy with some related projects. Chapter 3 overviews the Revy system, thus providing a background for understanding the following three chapters which discuss the architecture and design of the three subsystems of Revy: the Visualization and Interaction Subsystem (Chapter 4), the Parsing and Annotation Subsystem (Chapter 5), and the Runtime Modeling subsystems (Chapter 6). Chapter 7 presents experimental results, justifying the usefulness of Revy. Finally, Chapter 8 concludes the thesis, outlining several future research items and giving a summary.

Chapter 2

Related Works

The result of this thesis is the implementation of a tool, Revy, that visualizes the granularity and the runtime behaviour of a user's object-oriented program and provides hints for program parallelization. Although other research has addressed different aspects of Revy, we know of no other project that has the same goal. In this chapter, we examine several of these related projects, with emphasis on the three areas: visualization, granularity and program parallelization. This survey is not meant to be exhaustive, but rather it gives a flavor of what others have done.

2.1 Visualization

As pointed out in [TU94], program visualization research has been motivated by the desire to explain the functioning of algorithms by means of an animated display. Researchers usually apply visualization techniques for program understanding, testing and reverse engineering. In the course of this project, we have studied two visualization systems: Enterprise and Eiffel // [1].

2.1.1 Enterprise

Enterprise ([SS93], [IM95] and [MA96]) is an integrated parallel programming system that provides facilities to create, compile, execute, tune, and debug parallel programs on a network of workstations. The goal of the Enterprise project is to simplify the task

[1] Eiffel //, pronounced "Eiffel Par", stands for Eiffel Parallel.

of writing correct parallel programs in C. Enterprise provides two integrated models: a *programming model* and a *meta-programming model*. The programming model specifies the sequential semantics of the user code, and the meta-programming model allows a user to specify the parallelism in an application. The meta-programming model is a hierarchical combination of *assets*, which represents the user's program as a business enterprise. It is described by an asset graph, which is created in the Enterprise graphical user interface (GUI). The Enterprise GUI has the following features: (1) user specified parallelization techniques for the application by means of an asset graph, (2) animation of the execution of parallel programs, and (3) presentation of performance statistics of the parallel program using coloring, visual cues, charts and other symbolic representations [WI95].

There exist many fundamental differences between Enterprise and Revy. Enterprise handles a procedural language (C), while Revy deals with object-oriented languages (C++ and Java). Enterprise helps users write parallel programs, while Revy helps users in visualizing and analyzing granularity in sequential programs. Since Enterprise is an integrated parallel programming environment, it provides a user-friendly interface for: parallel programming by means of an asset graph, source code editing, compilation and execution, and performance tuning debugging using animation [IM95]. Revy provides users with an interface for: declaring a project, parsing source code, compilation and execution, building an object call graph, collecting runtime statistics about granularity, and identifying granularity by means of visualization and animation. The statistics gathered are for both object and messages in an OO program. Although the two projects serve different purposes, some of the concepts of Revy emerged from the Enterprise project.

2.1.2 Eiffel //

Eiffel // [AC96], is an extension to the programming language Eiffel. The Eiffel // system provides an interactive programming environment for parallel object-oriented programming and visualization. It provides features such as the graphical representation of objects, visualization of semantic rules of the Eiffel language, and animation

tools to show the concurrent activities of objects. The graphical environment displays objects and their interactions (object topology, attribute values, concurrent activities, subsystems, synchronizations), provides a set of primitives for controlling and probing the execution (granularity of interleaving, step-by-step execution, control over the interleaving), and detects deadlock configurations when they occur. The most interesting contribution of their system is the collection of runtime data without user instrumentation. Instead, they use semantic rules to express the dynamic behaviour of a program. They claim that their system contributes: (1) a pedagogic environment to demonstrate concepts of object-oriented programming and formal semantics, and (2) a step towards environments for the formal study of parallel object-oriented programming.

Eiffel // helps users investigate a parallel OO program using rich visualization support. The information gathering, execution control and performance tuning features provided are extensive. To collect the runtime behaviour of a user's program, they introduce an additional abstract layer, the semantic layer, which expresses the dynamic binding behaviour on a semantics engine. As a result, their system is language neutral since it needs "no instrumentation code". The user's program can therefore be kept semantically correct. Such abstraction means that no probe effects due to the execution of instrumented code are introduced. However, there are a few drawbacks to this approach. Defining the semantics for a language is not an easy task and is, in fact, an extremely abstract concept. As a result, the benefits of the language neutral feature are questionable. Moreover, the extra layer is subject to another kind of probe effect: the translation and communication between the extra layer. Unfortunately, the authors did not explain the process of semantic interpretation clearly, and provided the reader with no analysis on the performance of their system. For the area of visualization control, the Eiffel // environment is more elaborate than the Revy system. For instance, Revy provides no execution control, no inspection of objects' states and variables, and no source code animation since these features are not critical to Revy's intended goal. On the other hand, the layout of an object call graph in Eiffel // is too simple: there is no presentation of the objects' and messages' statistics, no features to unclutter call graphs, and so on. In essence, although Revy

and Eiffel // share the objective of visualizing OO programs, the exact goals are different: visualization of execution control versus visualization of granularity.

2.2 Granularity

Granularity is one of the key factors for efficient program parallelization. By understanding the granularity of their programs, developers can identify the potential parallel performance bottlenecks and decompose their programs more efficiently. Some tools already exist to help developers determine the granularity of the components of their programs. This section evaluates three existing tools for profiling sequential programs: `gprof`, "`java -prof`" and ObjectTrace.

2.2.1 Profiling Tools

The command `gprof` is a UNIX utility program for profiling C and C++ programs. It displays the call-graph profile data obtained by executing a program compiled with the "`-pg`" option. The report produced by `gprof` consists of a textual call graph, an execution time profile giving the CPU time for each routine itself and for the routines called by it, the number of times each routine was called, the fraction of the total time spent in it, and the average processing time per call. The "`-pg`" option causes a C/C++ compiler to instrument the user's code with the UNIX signal `SIGPROF` handlers and interrupts for logging function invocations by regular sampling. The log file is then interpreted by `gprof` to produce human-readable profiling information.

In Java, virtual machines (VMs) come with an option "`-prof`" which traces method calls in a Java program. Similar to the `gprof` command, "`java -prof`" provides the user with method-based profiling information, such as the method name, the number of times called, the method's caller and the processing time of the method. In addition, it provides minimal memory information such as the heap size and size of the free heap. Although different VMs may use different instrumentation techniques to implement this option, they do not modify the user's source code and the logging

is performed at a lower level – the interpreter level.

ObjectTrace [OS94] is a commercial tool from ObjectSoft Incorporated. It is a set of tools that can generate and interpret trace information for applications written in C++. ObjectTrace gathers statistics on: object creation, object destruction and invoked methods. It then builds a textual call graph from the trace data. It does not compute the memory used by an object or a method. However, it does report memory leaks (memory allocated for objects that is not released). ObjectTrace achieves this by instrumenting the C++ source code directly. Since the implementation details are proprietary information, it is difficult to comment on the efficiency and accuracy of the timings of their tool. The strength of ObjectTrace is its instance-based profiling data. For traditional profilers, the unit of operation is a function, while ObjectTrace keeps track of the lifetime and activities of all objects.

2.2.2 Critique

There are numerous advantages and disadvantages of the above tools as compared to the features of Revy. First of all, Revy is, except for the parsing and instrumentation part, a language neutral tool, while the rest of these tools are language specific. Second, these tools are inflexible in instrumentation. Users do not have full control on the procedures to be logged and presented. The "`java -prof`" command does not allow users to restrict the methods to be profiled. They can either profile all or none. In `gprof` and ObjectTrace, users can skip the profiling of all the methods in a source file, but not a single method. On the other hand, Revy allows users to selectively instrument each single method or whole classes. Third, the profiling information is not instance-based in traditional profilers nor in Java's built-in profiler. Instance-based profiling data is essential for programmers to understand and parallelize OO programs. This is due to the fact that different instances of the same class can execute methods for radically different time intervals, and different invocations of the same method can also vary considerably. Fourth, the data size of instances and messages, which are critical information for program parallelization, are not supported in any of these tools. The size of an object has to be taken into consideration when it has to

be transfered or replicated over a network. The performance of a Remote Procedure Call (RPC) is affected by the size of the parameters as they have to be packed at the local machine, sent and received over the network and unpacked at the remote machine. Fifth, as pointed out in [HU95], most of these tools attempt to measure the CPU time spent in executing the program; however, what the typical user today really cares about is clock time[2], not CPU time. Therefore, Revy computes the clock time spent on executing messages. Sixth, the measurement in `gprof` employs UNIX signals which may have a smaller probe effect, but uses regular sampling that is subject to statistical errors. On the other hand, Revy uses high-level instrumentation, which introduces higher overheads, but accurate statistics.

2.3 Program Parallelization

Researchers have been working on different approaches to transform a sequential program into a parallel one. One approach to program parallelization is to decompose the program into a set of communicating tasks and then apply some graph partitioning and task-to-processor mapping algorithms. The ideal approach would be a compiler that does all the work, for example, "`g++ -parallel myFile.cc`". In this section, we discuss two program parallelization projects: IRPC and `JAVAR`.

2.3.1 Parallelization Algorithms

One approach to program parallelization makes use of runtime performance data to determine the granularity and locality constraints of a program, and then applies graph partitioning heuristics to generate an optimal task-to-processor assignment. There have been some parallelization algorithms proposed, such as those presented in [CT92]: Bin Packing, Randomization, Pressure Model and the Manager-Worker scheme, and those discussed in [SB96]: Heaviest-Edge-First (HEF), Minimal Communication (MC), Kernighan-Lin (KL), and so on. This section presents a relatively

[2]The use of clock time measurement is valid only if the instrumented process does not compete with other processes for the resources such as CPU and I/O.

new approach, Inverse Remote Procedure Call (IRPC) [SB96].

IRPC is an integrated system with a compiler, a stub generator, and a runtime system. The runtime system executes a user program, collects the profiling data, builds an object call graph with a *cost model*, and applies the *IRPC heuristic* at runtime. The IRPC cost model measures: parameter passing costs, method invocation/return costs, execution costs, loop costs, and conditional split costs. The IRPC heuristic is based on the idea of "inverting" an RPC. That is, moving a procedure module transparently to the site of the caller. The heuristic uses a combination of execution and communication cost measurement with a finer granularity, by assigning objects or procedures to processors rather than assigning entire processes.

The authors tested their system with 27,000 artificial programs, and claimed that 95% of the tested programs obtained a traffic reduction cost over 75%. They also compared the results with the HEF, MC and KL algorithms. Overall speaking, IRPC is the only algorithm that has the lowest complexity as well as highest performance improvement for their class of tested programs.

The major advantage of IRPC is its consideration of objects. Many of the traditional parallelization algorithms employ a control flow analysis approach such as loop restructuring. However, when it comes to the object paradigm, data/object flow analysis plays a more important role. Moreover, IRPC illustrates the importance of instrumenting parameter sizes to calculate parameter passing costs. Our system Revy stresses the importance of objects and data sizes but it does not have any parallelization algorithms implemented yet.

2.3.2 Parallelizing Compilers

A parallelizing compiler takes a sequential program and automatically generates a parallel version while preserving the semantics of the original program. Parallelizing compilers are being researched at several institutions, for instance, the High Performance Java group from Indiana University is developing a tool called JAVAR [BG97].

JAVAR is a prototype Java restructuring compiler that can be used to make implicit parallelism in Java programs explicit by means of multi-threading. In particular, their research focuses on automatically exploiting implicit parallelism in loops and multi-way recursive methods. Parallel loops are either detected automatically using data dependence analysis, or are identified explicitly by the programmer's annotations. Because automatically detecting implicit parallelism in multi-way recursive methods can be very hard, they simply assume that such parallelism is always identified explicitly by means of annotations.

There are two main advantages of their approach. First, it makes a compiler handle the transformations that make implicit parallelism explicit, thereby simplifying the task of the programmer. Moreover, because parallelism is expressed in Java itself, the transformed program remains portable, and speedup can be obtained on any platform on which the Java Virtual Machine supports the true parallel execution of threads (for instance, the Java Native Threads package [SS97] from Sun). However, automatic parallelization is still a dream. Until it happens, we need tools like Revy to help us parallelize our programs.

2.4 Summary

This chapter presents various visualization, profiling and program parallelization systems, and contrasts them with Revy. To the best of our knowledge, none of the existing tools provides the integrated granularity computation and visualization environment that Revy supports.

Chapter 3

Revy System

Revy is a software system that aids users in instrumenting their source code, logging and collecting runtime traces of the execution results, and providing basic parallelization hints. Users interact with Revy using a Graphical User Interface (GUI). In this chapter, we give a brief discussion of the system requirements, its architecture and its design. This chapter provides the reader with an overview of Revy that will be elaborated in the following chapters.

3.1 Requirements

The prime requirement of Revy is to analyze the granularity of a user's application and provide hints for parallelization. This is achieved by:

1. parsing the user's source code,
2. inserting method calls into the code to compute message and object granularities,
3. compiling and executing the annotated code to gather trace and granularity information (such as instance size, message size and duration),
4. analyzing the traces and calculating runtime statistics to provide hints for parallelization, and
5. allowing the user to visualize the traces and statistics.

Revy uses a modular approach that divides these responsibilities into subsystems. The Parsing and Annotation Subsystem (PAS) is responsible for taking an object-oriented source program, parsing it and generating an instrumented version. The Runtime Modeling Subsystem (RMS) collects the runtime traces while the user's program is running. In addition, it establishes the object call graph from the traces, performs simple analysis and provides simple hints for program parallelization. The Visualization and Interaction Subsystem (VIS) is used for visualizing the program statistics, and providing an interface for users to interact with the other two subsystems and the operating system. The details of the requirements of each of these subsystems are presented in the following chapters.

3.2 Architecture

This section discusses the architecture of the Revy System. As mentioned before, Revy employs a modular approach. Such an approach allows users to interact with the subsystems directly (through shell commands), or to interact through a GUI such as VIS. The architecture is presented in the next two sections, and then the design is discussed in the following section.

3.3 Architecture Overview

Figure 3.1 presents the three subsystems and their interactions in brief. The left-hand side of the diagram shows the interactions between different subsystems using physical files, while the right-hand side shows the corresponding interactions using logical objects.

Users place their project details in a Revy Project File (RVY)[1]. The project file defines the source files, and the compilation and execution commands of the project. VIS is not only a graphical interface to the Revy system and various subsystems, but

[1]A user normally interacts with the Visualization and Interface Subsystem to create a project file. However, the user can also create the file manually.

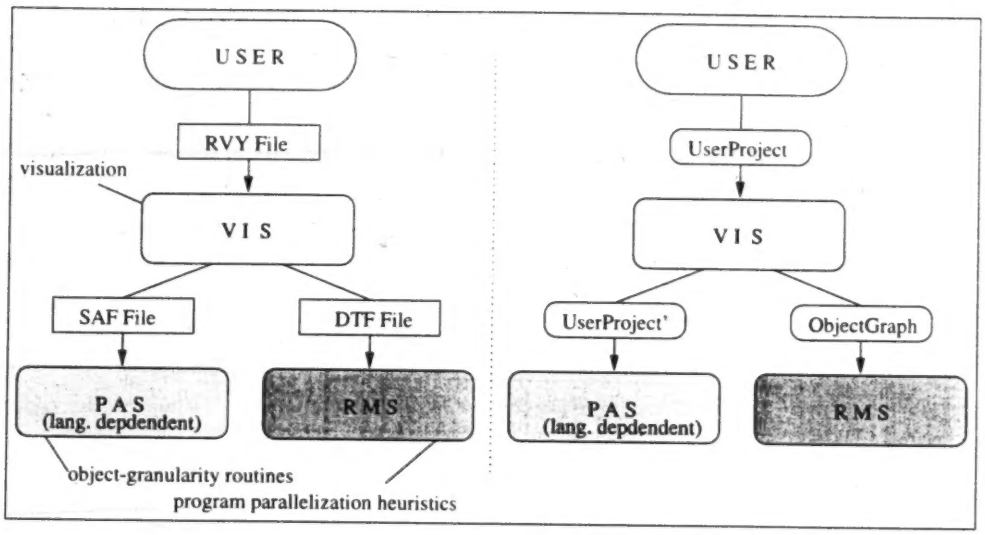


Figure 3.1: System Architecture in Brief

also a system that allows users to visualize their programs. VIS communicates with the Parsing and Annotation Subsystem (PAS) using a Static Artifacts File (SAF). PAS is the only language dependent subsystem. Separate versions of PAS for Java and C++ can easily be substituted. PAS is responsible for parsing and inserting granularity-logging routines in the user's program. The third component, the Runtime Modeling Subsystem (RMS) interacts with VIS via a Dynamic Trace File (DTF). RMS is used not only for collecting the granularity information of the program but also for calculating the runtime statistics.

The right-hand side of Figure 3.1 presents the logical representation of the physical files. The `UserProject` object, which represents the RVY file, contains the definitions of source files as well as the command strings to compile and execute the user's program. However, this is just the first incarnation of the object. As soon as the system parses the source files, the `UserProject` object also contains the classes and methods declared in the user's source files. This is the second incarnation of the `UserProject` object, and it is in fact the logical representation of the SAF file. The `ObjectGraph` instance is the call graph which represents the traces recorded in the DTF file.

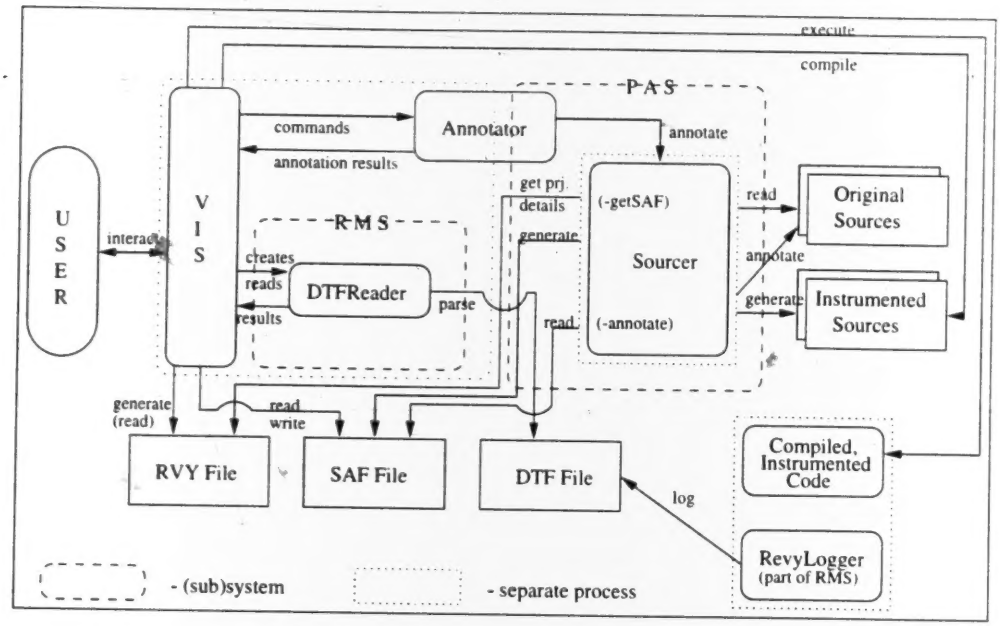


Figure 3.2: System Architecture in Detail

In brief, Revy uses a modular approach, with different subsystems targeted at different requirements of the Revy system. In a physical sense, they interact with each other through three files; in a logical sense, they interact via two objects. Such an approach enhances the maintainability and extensibility of the system.

3.4 Architecture Details

Figure 3.2 shows the detailed architecture of the Revy system. The Visualization and Interaction Subsystem serves as the interface gateway between the user and the subsystems. VIS displays object call graphs to users and allows them to interact with the graphs. In addition, VIS is responsible for executing simple operating system commands, for instance, compiling and executing the user's code.

The Parsing and Annotation Subsystem runs as a separate process. This is desirable since it is a switchable component of the Revy system. If the user's target language is not the default one[2], Revy can communicate with a different PAS process

[2]The default working language of Revy is Java.

for the specific source language. PAS receives commands (for example, to parse or to annotate) from Revy through an `Annotator` object that resides inside the Revy process. The `Annotator` object is in fact a wrapper object that hides the PAS subsystem from Revy. The details of the commands (for instance, the source files to parse and the classes to annotate) are stored in the RVY and SAF file, which are represented by a `UserProject` object.

The Runtime Modeling Subsystem resides inside the Revy process. The RMS contains a `DTFReader` object that is created by VIS. After parsing a DTF file, the `DTFReader` creates an `ObjectGraph` instance for VIS to display and for RMS to analyze. RMS can be started as a separate process. This is necessary when a GUI is absent and the user needs a textual presentation of the call graph and statistics. The `RevyLogger` object is considered as part of RMS, though it does not reside inside RMS. `RevyLogger` does not need to interact with the main Revy as it is merely a piece of code attached to the user's program for logging runtime traces that are used by RMS. The details of the architecture of the subsystems are presented in the next three chapters.

3.5 Design

This section discusses the design of Revy in brief by presenting two high-level diagrams: the data flow diagram and the state diagram. The data flow diagram shows how the input data is processed by the system until outputs are generated. The state diagram views the system using an event-flow approach. Along with the architectural figures, these diagrams provide a basic description of the Revy system.

3.5.1 The Data Flow Design

The data flow design describes the processing of inputs into outputs. In Revy, the inputs are the user's source files and the instrumentation directions for the code modules. Figure 3.3 shows the data flow design of Revy. Note that the large boxes denote

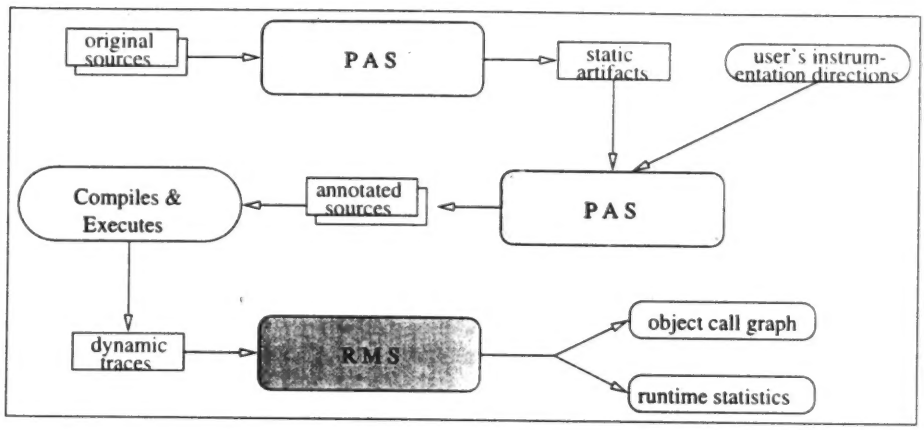


Figure 3.3: Data Flow Diagram of Revy

the processing units and the small boxes are the data to be processed. First, the user's source code is passed to the Parsing and Annotation Subsystem for parsing. PAS then analyzes the files and produces the static artifacts about the source code. Based on the user's instrumentation directions, PAS annotates the source code. The modified source code is compiled and executed, thereby generating dynamic traces. RMS takes the trace file, translates it and produces the outputs of the system: a logical representation of the call graph and the runtime statistics.

3.5.2 The State Design

The state design in Figure 3.4 shows the four states of the Revy system, and the edges are the events and user commands. The system starts in the "Empty Project" state. In this state, the user can either define a new project or open an existing one. As soon as the user has defined a new project, the system enters its second state: "New Project Defined", where it waits for the user's commands to instrument the code. When this command is given, the system proceeds to the third state: "Code Instrumented". In this state, the system waits for the compilation and execution commands, which lead to the generation of runtime traces. After generating these traces, the system enters the "Visualization of Object Call Graph" state. In this state, users can either query the runtime statistics for parallelization hints or refine their annotation directions so that the information gathered is more precise.

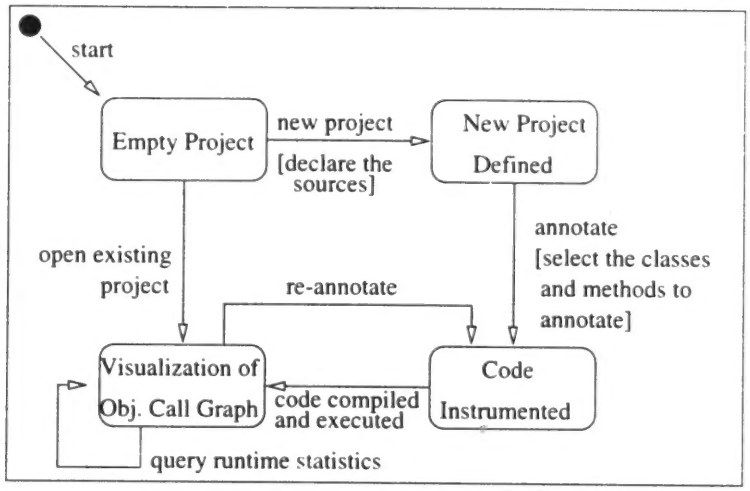


Figure 3.4: State Diagram of Revy

3.6 Implementation

The implementation of the major components of Revy uses Java version 1.1, on a Solaris 2.5 platform. Since Java is a "write-once-run-anywhere" programming language, Revy also runs on other UNIX boxes, PCs, or MacIntoshes. The Parsing and Annotation Subsystem for C++, called `C++Sourcer`, was implemented with the Solaris C++ compiler. The `C++Sourcer` requires that the Sage++ [ER95] library be installed. On the other hand, PAS for Java (i.e. `JavaSourcer`) uses the JavaCC [ST97] library. The Revy system does not have any specific hardware requirements.

3.7 Summary

This chapter states the core requirements of the Revy system - parsing, instrumentation, collection of program granularity, computation of the runtime statistics, and visualization. In addition, the architecture of the system is discussed. Finally, the data flow and the state design diagrams are presented. This chapter provides the overview needed to understand the following three chapters about the subsystems.

Chapter 4

Visualization and Interaction Subsystem

The Visualization and Interaction Subsystem (VIS) is the most important part of Revy to the user. It is not only the GUI of Revy but also the subsystem in which program visualization techniques are explored. In this chapter, we state the requirements of VIS, describe the visualization techniques implemented, provide a walk-through of its functionality, and suggest a few enhancements.

4.1 Requirements

VIS is responsible for providing a user-friendly interface for users to interact with the other two subsystems and the operating system. The requirements of VIS are:

1. specifying the source files for PAS to parse,
2. selecting the classes and methods for PAS to instrument,
3. compiling and executing the user's program,
4. building an object call graph from the runtime traces,
5. calculating various trace statistics and parallelization hints, and
6. visualizing object relationships and call patterns.

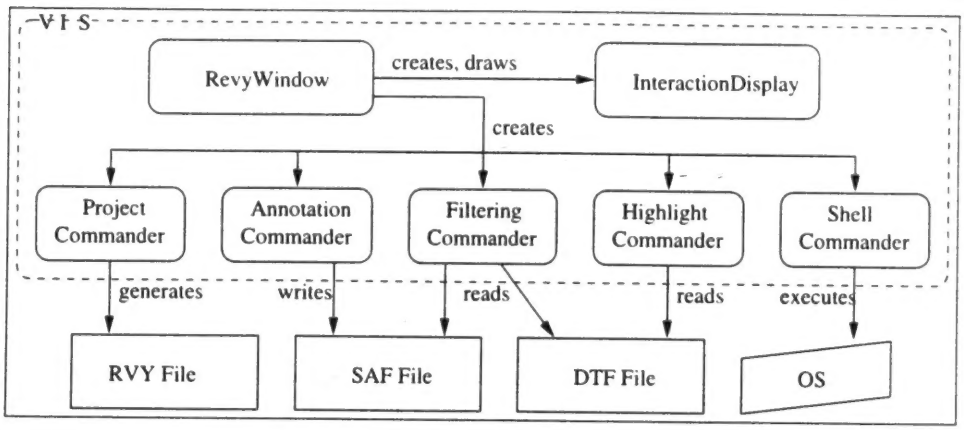


Figure 4.1: VIS Architecture

VIS employs some basic visualization techniques such as coloring, interactive dragging, and animation. Advanced techniques such as filtering and highlighting are used as well. Filtering of instances and messages is necessary since call graphs are often cluttered and difficult to analyze by human eyes. Highlighting gives the user a more noticeable view of an important object (for example, an object that should be parallelized). Besides, VIS provides miscellaneous functions such as instance and message inspections, a source file viewer, and so on.

4.2 Architecture

The detailed architecture of VIS in Figure 4.1 shows seven Abstract Window Type (AWT) objects: one main window, one canvas and five dialog windows. `RevyWindow` is the main window of the whole Revy system. The `InteractionDisplay` canvas displays the object call graph. Each of the five dialog windows (`Commanders`) deals with one of the three Revy files or the operating system. For example, `ProjectCommander`, `AnnotationCommander`, and `HighlightCommander` manipulate RVY, SAF and DTF files, respectively, and `FilteringCommander` deals with not only the SAF file but also the DTF file. `ShellCommander` sends shell commands to the operating system.

4.3 Design

This section covers the user interface (UI) design of VIS. UI design is human-machine interaction analysis/design. Before presenting the results of the analysis, we state the standards that the Revy UI design follows:

- Top-left to bottom-right: widgets are placed according to the order that the user will interact with them, from the top-left to the bottom-right corners of a window. This will minimize the mouse and cursor movements. For example, an Open Dialog has the directory list widget placed on the left, the file name list widget on the right, and the `OK` and the `Cancel` buttons at the bottom right.

- Proper use of colors: bright colors denote important objects. The user will therefore be able to identify the important objects on the display easily. This is especially helpful when the display is cluttered.

- Consistent widget placement: widgets are consistently placed on dialog windows. For example, a Save Dialog looks the same as an Open Dialog, except that the Save Dialog has an extra widget – the file name field. Therefore, the common widgets (directory list, file name list and the buttons) are located at the same positions on both dialogs. As a result of this standard, a reusable template dialog window called "`Commander`" was implemented.

The process of UI analysis/design follows the Task Analysis methodology [GH97]. Task Analysis describes a range of procedures, the aim of which is to document the actions performed by a user when interacting with a computer system. For the Revy UI design, the high level actions are documented. The documentation begins with a "fresh" Revy where the user has a sequential program only, and ends when he or she obtains the knowledge on how to parallelize the application.

The sequence of a Revy user's tasks are as follows: (1) starting the Revy system, (2) defining a new project, (3) directing the system to annotate the code, (4) compiling and executing the program to obtain runtime traces of the program, and (5) viewing the object call graph. After looking at the call graph, the user may want

to: (6) inspect how the program runs (that is, the call sequence), (7) filter some of the data if the call graph is too cluttered, (8) highlight the most active instances and messages, or (9) investigate the runtime statistics of the application to obtain clues for parallelization. All these steps together with the relevant screen shots are presented in Section 4.4.

The UI design process resulted in several AWT classes. We present the Class, Responsibility and Collaborator (CRC) cards in the appendices. Appendix G shows the supporting classes for VIS. The supporting classes are the AWT classes and View classes. Views are components of the Model-View-Controller (MVC) paradigm [KP89]. In our case, the View classes are views on the message and instance models. The Model classes will be discussed in Section 6. Readers may want to study the models before investigating the View classes. Appendix H presents the major classes of VIS, including `RevyWindow` and the dialog window classes.

4.4 Revy Graphical User Interface

In this section, a walk-through of the Revy user interface is presented. It begins by discussing how to start the Revy system. It follows the normal steps of the user, ending with parallelization suggestions for the user.

4.4.1 Revy Main Window

Before starting the Revy system, a user must define the `REVY_HOME` environment parameter in a "`.revy`" file. The file should be placed in the current working directory, otherwise Revy assumes the user's home directory has a subdirectory called "`revy`" which is the default Revy home directory. The user can execute the command "`revy`" to start the Revy system.

The Revy main window is composed of four sections: the Menu Bar, the Button Panel, the Main Display and the Status Panel (Figure 4.2). The Menu Bar is self-

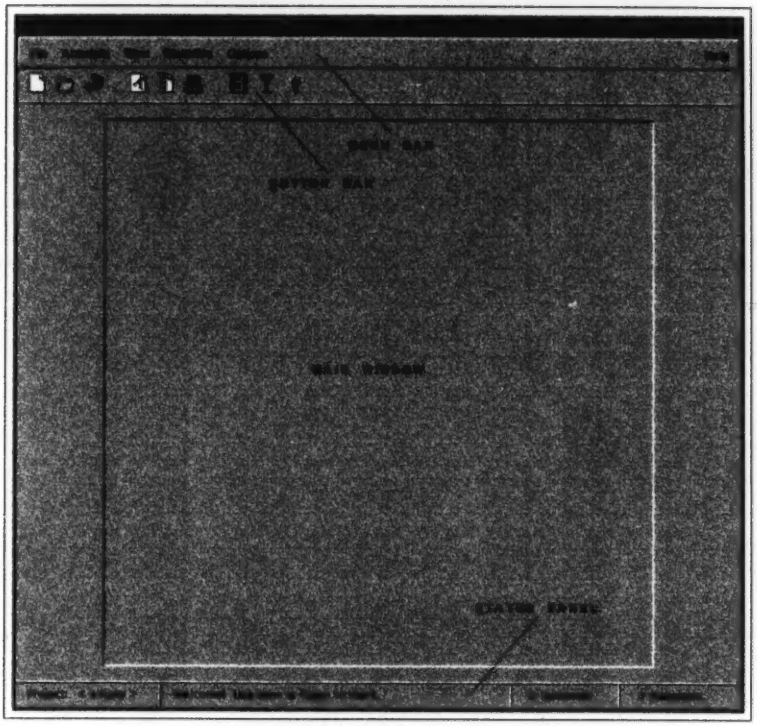


Figure 4.2: Revy Window

explanatory. The Button Panel consists of nine icon buttons whose picture may not seem obvious at first glance. However, users can find out the meaning (tooltips) of a button by pointing at it and looking at the description in the Status Panel. Figure 4.3 lists the description of the buttons. The Main Display is used for displaying an object call graph, or in the future, displaying the class diagram as well. The Status Panel shows which project the system is dealing with, the messages of the system to the user, and the number of instances and messages discovered in the runtime traces.

4.4.2 Defining a New Project

To define a new project, users select the `New Project` menu item or the new project icon button. The system will pop-up a dialog window (Figure 4.4) prompting the user for various parameters of the new project, such as the project name and the home directory of the project. The user can click on the arrow buttons to add or remove

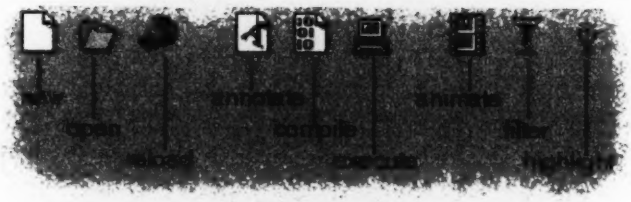


Figure 4.3: Button Panel

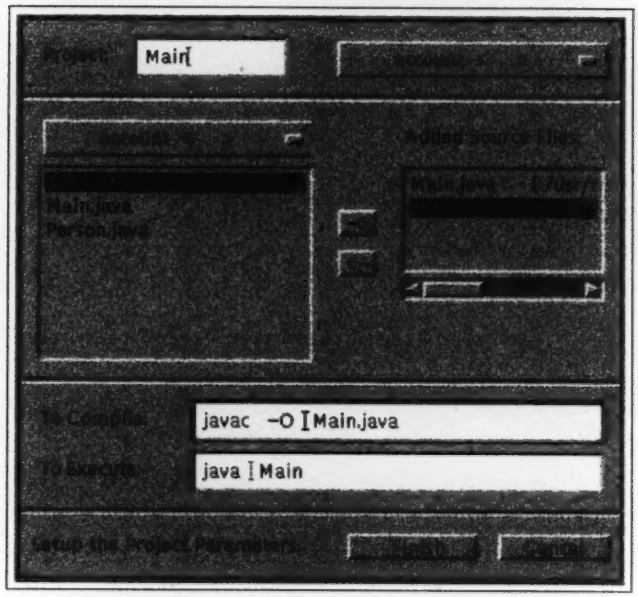


Figure 4.4: Project Dialog Window

the source files of the project. At the bottom of the dialog are the compilation and the execution command strings. These strings are automatically generated[1] for the user. Extra flags, for example the "-O" flag for optimization, can be added in the command strings.

4.4.3 Annotating User's Code

Revy will generate a project file ("`.rvy`") immediately after the user has defined the project. The file is human-readable, and it contains the names of all the source files and the two command strings. Revy will then invoke the Parsing and Annotation

[1]The defaults are Java compile and execution commands.

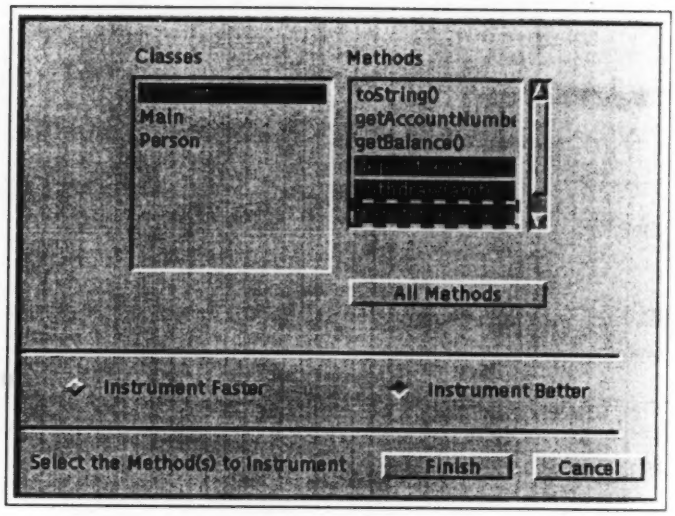


Figure 4.5: Annotation Dialog Window

Subsystem to parse the source files and generate a Static Artifacts File ("`.saf`"). This file contains all the names of the classes and methods defined in the source files. Revy will inform the user through the Status Panel of the number of classes and methods which have been successfully identified[2].

The user can then click on the annotate icon button or select the **`Annotate Code`** menu item to direct the system to instrument. Figure 4.5 shows the dialog window for annotation. The user first selects the class, and then picks the methods to instrument. The user can also click on the **`Select All`** button to choose all methods of that particular class. Since different selection mechanisms are provided, the annotation process is more flexible than other profiling tools. At first, users will want to instrument all methods. Then, as they iterate through their analysis, they will likely be more selective in what they want to profile.

After the class and method selections, the next step is to select either the **`Faster`** or the **`Better`** instrumentation methods. The difference between these two instrumentation methods will be described in Section 5.5.3. Finally, the user can click on the **`Finish`** button to initiate the instrumentation process.

[2]Only concrete (non-abstract) classes and public instance methods are reported.

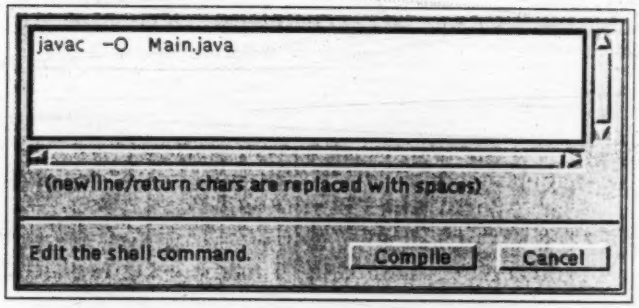


Figure 4.6: Compile Dialog Window

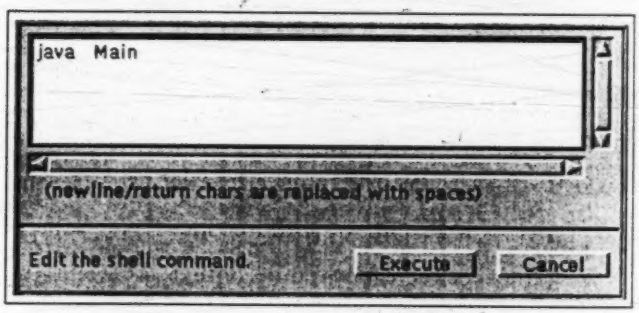


Figure 4.7: Execute Dialog Window

4.4.4 Compiling and Executing

Once PAS finishes instrumenting the program, Revy will notify the user about the number of classes and methods that have been annotated. The user can then compile and execute the program by clicking the compile and execute icon buttons. These two buttons generate two dialog boxes, as shown in Figure 4.6 and Figure 4.7, respectively. The boxes contain editing areas in which the user can modify the shell commands. Note that Revy will display the results of the compilation and execution processes on the Status Panel. Revy assumes a DTF file is generated once the execution completes successfully. After that, it will read in the runtime traces and display the object call graph on the Main Display.

4.4.5 Visualizing the Runtime Object Call Graph

The runtime object call graph is displayed on the Main Display. As we can see from Figure 4.8, circles (nodes) represent instances and directed lines (edges) represent messages. The numbers beside the names of instances and messages are the object and message IDs, respectively. These IDs are system generated, and are used to differentiate instances/messages of the same name. To reduce the cluttering problems, three techniques are applied to produce a call graph with better visibility. First, instance and message names are abbreviated to eight characters long. Second, message names are placed randomly, but near the middle of each edge. Third, instances are displayed in a circular pattern. In other words, the `main` (or `root`) instance starts from the center, with instances of object IDs 1 to 4 surrounding it. The instances of object IDs 5 to 12 surround the inner circle, and so on. This circular display is similar to the Hyperbolic Display in [LR95]. The Hyperbolic Display, which "supports a smooth blending between focus and context, as well as continuous redirection of the focus", arranges all the leaf nodes along the circumference of a circular display region. It uses tiny circles for the nodes so that all of them can be displayed on one plane. In our case, more information has to be displayed, such as the instance and message names. Therefore, the nodes are drawn larger to fit in the names. In the future, a true Hyperbolic Display could be implemented as a zoomed-out mode of Revy.

Before discussing the details of the display of the call graph, we define the *active time* of an *object*. In Revy, the term object can refer either to an instance object or a message object. Active time has a different meaning for each kind of object. For messages, active time is the processing time, excluding overhead times such as method invocation time and the time to return results. In contrast, the active time of an instance object is its execution time; that is, the total of the active times of all its messages. We define the activity level of a message to be its active time divided by the system's total message active time, while the activity level of an instance is its active time divided by the system's total instance active time.

The main display has four features which are not so obvious. First, the color of a

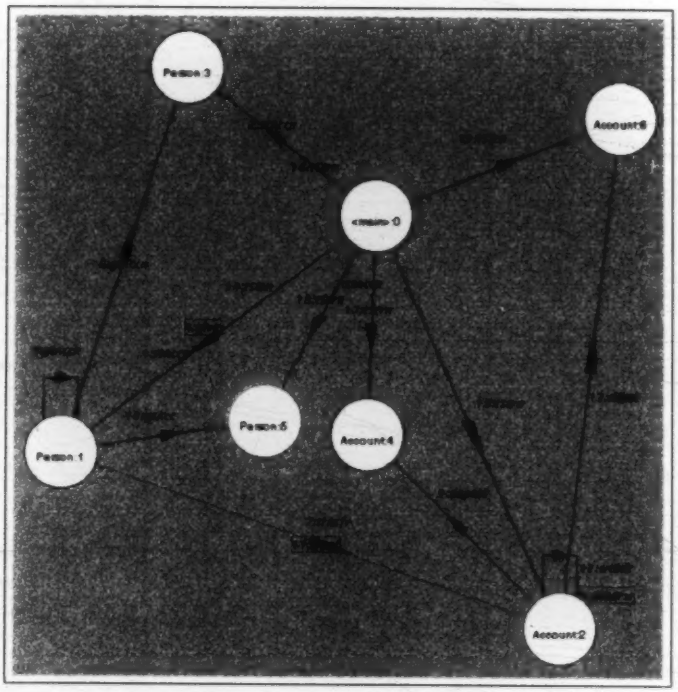


Figure 4.8: Visualizing an Object Call Graph

node or an edge identifies its activity level. Red is used for any object with an activity level over 60%, blue is used for levels over 30% but less than 60%, green is used for levels less than 30% and over 10%, and black is used for levels less than 10%. Second, the details about each object can be viewed by clicking the right mouse button on the object. Revy then pops up an Inspector Window which shows the object's name, class, ID, active time, activity level, and so on. Third, message names can be hidden by clicking the left mouse button. A click on an edge hides the name of the message; a click on a node hides the names of all its messages. To hide all message names in the graph, the user selects the **Hide Message Names** menu item. Fourth, the edge names and the nodes are draggable. As a node is being dragged, the edges leading to it are moved accordingly. These last two features help users unclutter their object call graphs.

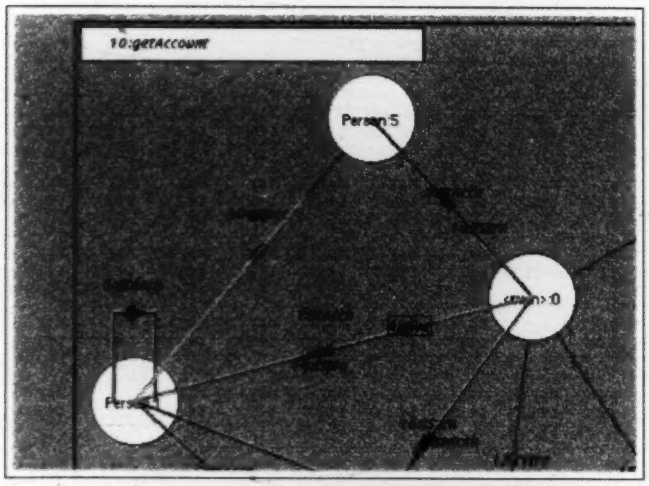


Figure 4.9: Animating Execution

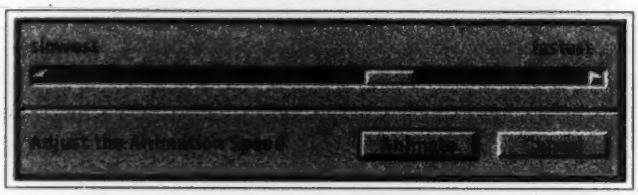


Figure 4.10: Animation Dialog Window

4.4.6 Animating Execution

There are three steps to obtain a good visualization of the runtime behaviour of a program. They are: (1) animating program execution, (2) filtering the less important objects from the graph, and (3) highlighting the most important objects in the graph.

The `Code Execution` menu item allows users to view an animation of how the program runs as recorded in the runtime trace file. With this feature, users can gain a better understanding of the calling patterns and the sequence of messages. While the animation is running, the name and the order of the active message is shown in the top-left corner of the Main Display, and the message is highlighted to denote that it is active (Figure 4.9). The speed of the animation can be adjusted with the scroll-bar of the Animation Dialog (Figure 4.10). Note that a runtime trace records only one instance of the execution. Programs may have a totally different execution

pattern on another run, depending on the input and environmental parameters.

4.4.7 Filtering Instances and Messages

Filtering helps users unclutter the object call graph by hiding less important objects. For example, the user may want to filter all the "get" and "set" messages, as well as all basic instances (such as something simple like a `PhoneNumber` instance). With the Filtering Dialog, as shown in Figure 4.11, users can filter a single message, all messages sent and received by an instance, all invocations of a method, and all instances of a class. Note that an instance will not be displayed if it does not have any displayable messages. In other words, there are two scenarios in which an instance is not shown: (1) an instance does not have any recorded messages sent or received, or (2) the instance has all its messages filtered. The filtering function is accessible with the `Filtering` menu item or the filter icon button.

4.4.8 Highlighting Instances and Messages

While filtering unclutters a call graph by hiding unimportant objects, highlighting makes an important object more noticeable by displaying it with a bright color. A user can access the highlight dialog window (Figure 4.12) with the `Highlight` menu item or the highlight icon button. As shown in the figure, Revy can highlight any instance or message whose active time, activity level or the rank of its activity level fulfills the criteria set by the user. The final highlighted objects are based on the "or" result of the criteria. For example, as shown in the figure, Revy will select any instance whose rank is in the top three, and any message whose processing time is greater than 5000 milliseconds *or* whose activity level is greater than 30%. Revy will then draw those selected messages with a light yellow background and inform the user of the number of highlighted instances and messages through the Status Panel.

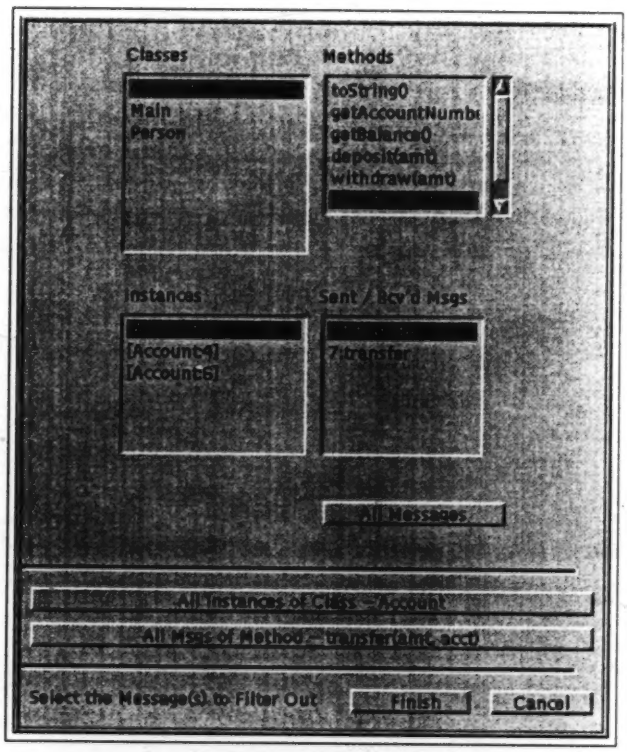


Figure 4.11: Filtering Dialog Window

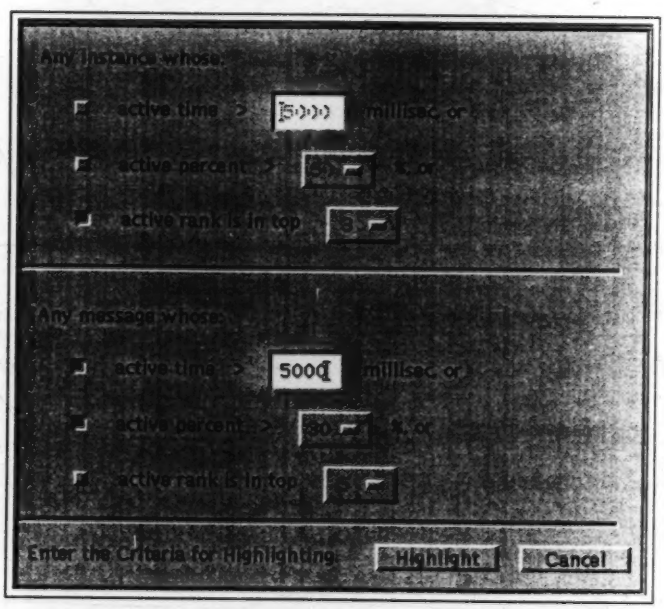


Figure 4.12: Highlight Dialog Window

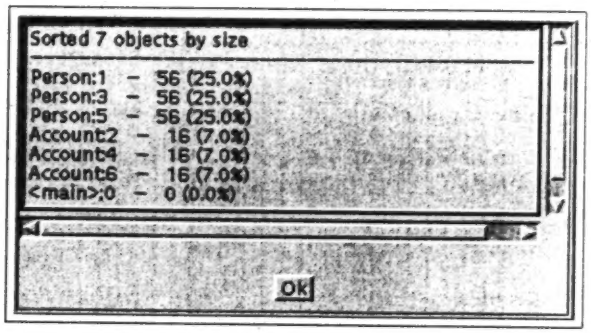


Figure 4.13: Displaying Runtime Statistics

4.4.9 Displaying Runtime Statistics

Users are often most interested in the runtime statistics of an application. They can query the Revy system for various statistics from the `Statistics` menu. Figure 4.13 displays an example of the statistics a user can obtain. Revy processes and sorts the data before presenting it to the user.

Before discussing the supported runtime statistics, we elaborate our definition of message active time. Recall from Section 4.4.5 that the active time of a message is the processing time, excluding any overheads. In fact, message active time refers to the *net* processing time, and aggregate message active time refers to the *total* processing time of a message. For example, assume there is a message `transfer()` which sends two internal messages, namely `withdraw()` and `deposit()`. The aggregate message active time of the `transfer()` message is the difference between the recorded start and finish times of the message. Its (net) message active time is its aggregate active time less the aggregate active times of the `withdraw()` and `deposit()` messages.

In the current implementation, users are able to view the following trace statistics:

- *Sort Instances by Object ID* – the IDs of all instance objects,
- *Sort Instances by Size* – the size[3] of all instance objects,
- *Sort Instances by Activity* – the activity level of all instance objects,

[3]Instance size is measured in bytes.

- *Sort Instances by Average Activity* – the average activity level of instance objects grouped by class (in other words, instances are grouped by their classes first, then the average activity level is calculated),
- *Sort Instances by Adjacency* – the number of neighbours (instances) of all instance objects,
- *Sort Messages by Message Order* – the IDs of all messages[4],
- *Sort Messages by Size* – the size of all messages[5],
- *Sort Messages by Activity* – the activity level of all messages,
- *Sort Messages by Average Activity* – the average activity level of messages grouped by methods (in other words, messages are grouped by their method signatures first, then the average activity level is computed), and
- *Sort Messages by Aggregate Activity* – the aggregate activity level of all messages.

Average Activity is the average activity level of the instances/messages grouped by the corresponding aspects. Averages are necessary because we have noticed, during our research, that the performance of some methods varies drastically depending on the states of the objects and the program. One good example to illustrate the problem is the methods of a `BinaryTree` class. Suppose there is a method `search()` defined for the class. Since trees are nested in nature, one can easily see that the searching of the median number runs fast; but the searchings of the two extreme-most numbers take longer times to run, depending on the height of the tree.

4.5 Further Enhancements

Visualization of call graphs plays a very important role in program understanding. Users want to perceive more information at one time while maintaining a high read-

[4]Message IDs are in the order of the start times of the messages.

[5]Message size is measured in bytes, and defined as the total size of all the parameters of the message.

ability of the display. The next evolution of the Visualization and Interaction Subsystem should include some features such as grouping instances and messages, zooming in/out, auto-filtering of basic instances and messages (such as `PhoneNumber` objects and get/set messages), source code animation [AC96], and so on. We can also consider implementing the Hyperbolic Display technique as described in [LR95] and [MB95]. In essence, the key problem in program visualization is to reduce the amount of clutter in the display. Suggestions for better visualization of programs with large hierarchies and complex topologies can be found in [KS93], [TU94], [LR95], and [MB95].

4.6 Summary

This chapter discusses the Visualization and Interaction Subsystem in depth: its requirements, architecture and design. Moreover, it presents the different features of the Revy GUI and provides a detailed walk-through on how a user can start from a sequential program to the point where he or she acquires the knowledge on how to turn it into a parallel application. Finally, this chapter suggests a few aspects for further enhancing the visualization in VIS.

Chapter 5

Parsing and Annotation Subsystem

The Parsing and Annotation Subsystem (PAS) of the Revy system instruments the user's source code so that information is gathered and saved to a trace file as the user's application is running. In this chapter, we list the requirements of PAS, present its architecture, outline its design, describe its implementation, discuss some interesting problems and list a few future enhancements.

5.1 Requirements

The Parsing and Annotation Subsystem takes an object-oriented source program as input. The output of PAS is an instrumented version of the user code that is semantically the same as the original. The user can direct PAS via the Visualization and Interaction Subsystem (VIS) to instrument any classes and methods. To achieve this, PAS generates an intermediate output which is the static artifacts of the user code. As the user's program is running, it generates runtime traces that can be used by the Revy Runtime Modeling Subsystem (RMS) to analyze the object communication patterns of the program. The trace file, which is written by the PAS-instrumented code, contains the following information:

- creation of each object,
- destruction of each object [1],

[1] Only for languages that have explicit destructors, like C++.

```
public void transfer(double amt, Account acct)
{ /* Original Code */
  this.withdraw(amt) ;
  acct.deposit(amt) ;
}
```

```
public void transfer(double amt, Account acct)
{ /* Instrumented Code */
  int _sze = 0 ; /* size of the message */
  Hashtable _vset = new Hashtable(32) ;
  _sze += 8 ; /* double */
  _sze += (acct==null)?  _NULL_SZE:acct._rvRuntimeSize(_vset) ;
  _RvLogger._rvLogStart(this, "transfer",
    "!double^0!Account^0!", _sze) ;
  {
    this.withdraw(amt) ;
    acct.deposit(amt) ;
  }
  _RvLogger._rvLogEnd(this, "transfer", "!double^0!Account^0!") ;
}
```

Figure 5.1: Method Annotation

- size of each object [2], and
- for each method invocation, the start and end times, the sender and receiver objects, the message size and the method signature.

Suppose we have the method **transfer** which is a member of the class **Account**. Figure 5.1 shows how the method is instrumented in order to collect the trace information stated above. Note that the objects, messages and variables that start with underscore ("_") are the code generated by PAS. The statement "**_rvLogStart()**" directs the system to log the fact that the object **this** is executing the **transfer** method (the first and second arguments of the statement). The timing of the message is logged by PAS internally. The identifier of this object can be retrieved by sending instance methods to the **this** object. The identifier of the **transfer** method

[2]Both static and runtime sizes are supported.

```
<Msg cls=Account oid=6 mth=transfer sgn=!double^0!Account^0!
  sze=24 tid=0 t=308>
</Msg cls=Account oid=6 mth=transfer sgn=!double^0!Account^0!
  tid=0 t=313>
```

Figure 5.2: Trace Output

(that is, the method signature) is recorded by PAS as the third argument. The argument `_sze` tells the system the total message size – the size of the parameter `amt` (double, 8 bytes) plus the runtime size of the object `acct`. Finally, the statement "`_rvLogEnd()`" informs PAS that the invocation of the message `transfer` upon `this` object is now finished. The actual trace output of the above code, which consists of one method start log and one method end log, is presented in Figure 5.2.

The PAS requirements are independent of any specific object-oriented language. The primary goal of this project is to maintain this language independence as much as possible throughout the instrumentation process. However, there are critical differences between various OO languages, and these differences introduce interesting design and implementation challenges. PAS has been implemented for two languages, Java and C++, and can easily be extended for other OO languages. The secondary goal is to produce a solution that is independent of the target machine. That is, Revy should be portable across different OS platforms.

5.2 Architecture

This section presents the architecture of PAS, by describing its major components and how they interact within PAS as well as with other parts of Revy. As pointed out in [KS93], there are various ways of collecting program dynamics, such as special hardware, software instrumentation, and alternative methods like monitoring tools. Revy employs software instrumentation techniques for PAS.

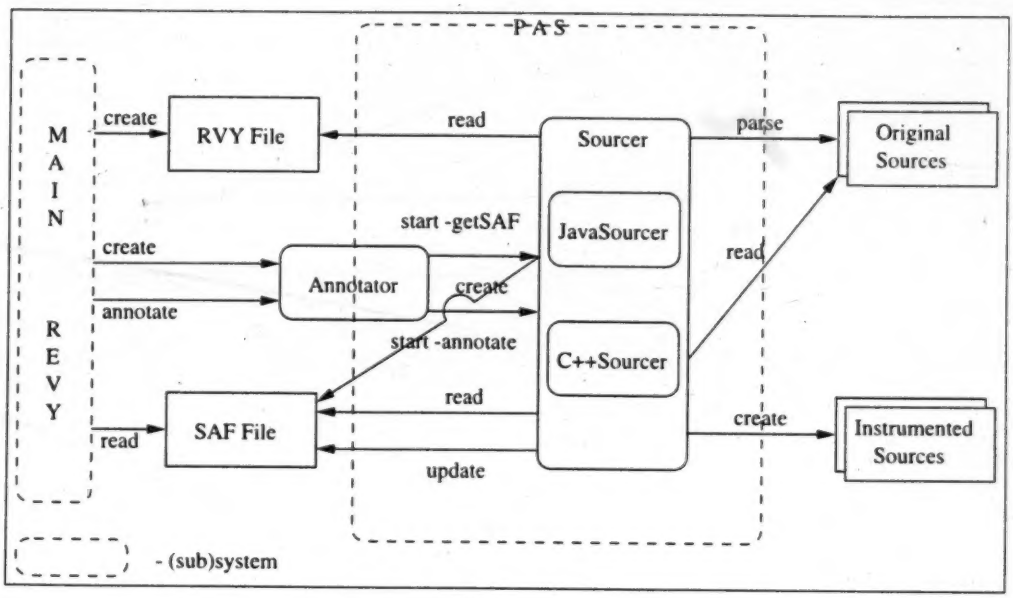


Figure 5.3: PAS Architecture

5.2.1 PAS Architecture

The major objects of PAS are the `Annotator`, `JavaSourcer`, and `C++Sourcer`. Figure 5.3 shows the software architecture that implements PAS. From the figure, we can see that `Annotator` mainly operates as a service distributor [3] on behalf of `JavaSourcer` and `C++Sourcer`. It receives requests from the main Revy system and routes the request to one of the Sourcers depending on the source language. The Sourcers are responsible for parsing, collecting static artifacts and inserting instrumentation code into user's source files.

`Annotator` is the only PAS object which interfaces with the rest of Revy, and it is responsible for starting the Sourcers. The Sourcers produce outputs which are read by the main Revy system. As shown in Figure 3.2, Sourcers run as separate processes. In fact, they are executed in two passes. The first Sourcer process parses and extracts static artifacts in response to the "-getSAF" flag. Then, in response to the "-annotateBetter" or "-annotateFaster" flags, the second Sourcer process inserts instrumentation code according to the user's direction. The details about these two

[3] The design pattern [GH94] name is Strategy.

flags are discussed in Section 5.5.3. There is a fourth flag understood by the Sourcers. It is the "-restore" flag, which restores the user's original source files. All these flags are transparent to the users if they are using the GUI. However, they have to use the flags if they start the Sourcers on command lines.

5.2.2 Architectural Strategies

During the analysis phase of PAS, we identified four different strategies for software instrumentation of a user's object-oriented program:

1. Write a source-to-source translator to insert instrumentation source code into the user's code.

2. Write an object-code-to-object-code translator to insert instrumentation object code into the user's byte code (Java) or machine code (C++).

3. In the case of Java, adapt the existing "-prof" flag of the Java Virtual Machine (VM) for profiling a Java program.

4. Modify a Java byte code interpreter to produce a trace file.

Of the four options, the first was chosen. The second option was rejected for four reasons. First of all, it is just as difficult to translate object code as it is to translate source code. Moreover, it is more difficult to debug object code translators since neither the inputs nor the outputs are human readable. Furthermore, this solution is not language independent since byte code translation retains portability, while machine code translation does not. In other words, for Java, this option produces a solution that is portable, while for C++ each target OS needs its own translator. Finally, with regards to optimization, the modified object code may violate the optimizations already introduced, and the inserted instrumentation code is not optimized at all.

The third and fourth options were rejected for a few reasons. The third one does not meet the requirements stated earlier since the "-prof" flag does not give us enough information (Section 2.2.1), unless we elaborate it by doing something like what the fourth option does. The fourth option is poor as it is not language independent. It

only works for Java, but in the case of C++, there is no interpreter to modify for compiled C++ code. Besides, even for those OO languages like Java that use interpreted byte codes, each target platform needs its own version of interpreter, so the interpreter modifications are not portable.

The result of implementing the first option is a source-to-source translator named Sourcer. We have built two Sourcers: C++Sourcer and JavaSourcer. The C++Sourcer was written first, in C++ using Sun's CC compiler and the Sage++ Compiler Toolkit version 1.9 [ER95] from the Extreme Research Group of Indiana University. The JavaSourcer was written in JDK version 1.1 [JS97] and JavaCC version 0.6.1 [ST97] with the standard Java language grammar [GJ96]. JavaCC, which stands for Java Compiler Compiler, is from SunTest (a spin-off from Sun Microsystems)[4].

5.3 Design

This section describes the major objects of PAS, their interaction and the text files that serve as the interfaces among them.

5.3.1 Interface Files

Two files are used to transfer information between the components of the main Revy system and PAS: the Revy Project File (RVY) and the Static Artifacts File (SAF). Both of these files are stored as text. There is one RVY file for each user project (application). A RVY file can be specified by programmers via VIS or with any text editor. The file contains the names of all source code files used in the project as well as the compile and execute commands. A SAF file, which is produced by a Sourcer in its first pass, contains the project name, source file names, type table, and the class, field and method tables for the user's source code. A programmer uses VIS to select which classes and methods to annotate. These selections are stored in the SAF file. Another Sourcer process, which reads in the SAF file in its second pass, annotates

[4]We could also use JavaCC to build the C++Sourcer since it comes with a C++ grammar specification. This would reduce system maintenance since both Sourcers would be similar.

the user's code. Note that a SAF file is in an HTML-like format which allows the file to be easily viewed with a modified Web browser. Samples of the interface files are given in Appendices D and E.

5.3.2 Class, Responsibility and Collaborators

This section describes all the PAS classes in detail by providing CRC cards for each of them. Appendix I shows the supporting classes used to model the user's classes, fields, methods and types. Appendix J presents the major objects involved in the instrumentation and logging processes. Note that the `JavaSourcer` class inherits the template code of a Java parser from the JavaCC package. Since the template code was written in a procedural way, the `JavaSourcer` also follows this convention. The details of the C++Sourcer are omitted since they are similar.

5.3.3 Collaboration Diagrams

Two high-level collaboration diagrams outline the responsibilities of PAS. These two diagrams constitute the two passes of the Sourcers. The first pass has two responsibilities: parsing and extracting static artifacts. Figure 5.4 shows how static artifacts are collected from the user's source code. It first parses concrete classes, instance or static fields, instance methods, and parameters of each method, and records them as the corresponding `UserCode` objects. Each `UserCode` object is then directed to write out the static artifacts into a SAF file. The second pass is to insert instrumentation code. Figure 5.5 demonstrates how a user source method is annotated. It first reads the SAF file which contains the user's directions on instrumentation. It then gets the method signature and looks up the annotation direction for a method with that signature. As it reads the tokens in the method body, it inserts the method-start and method-end code and generates the output into a new source file.

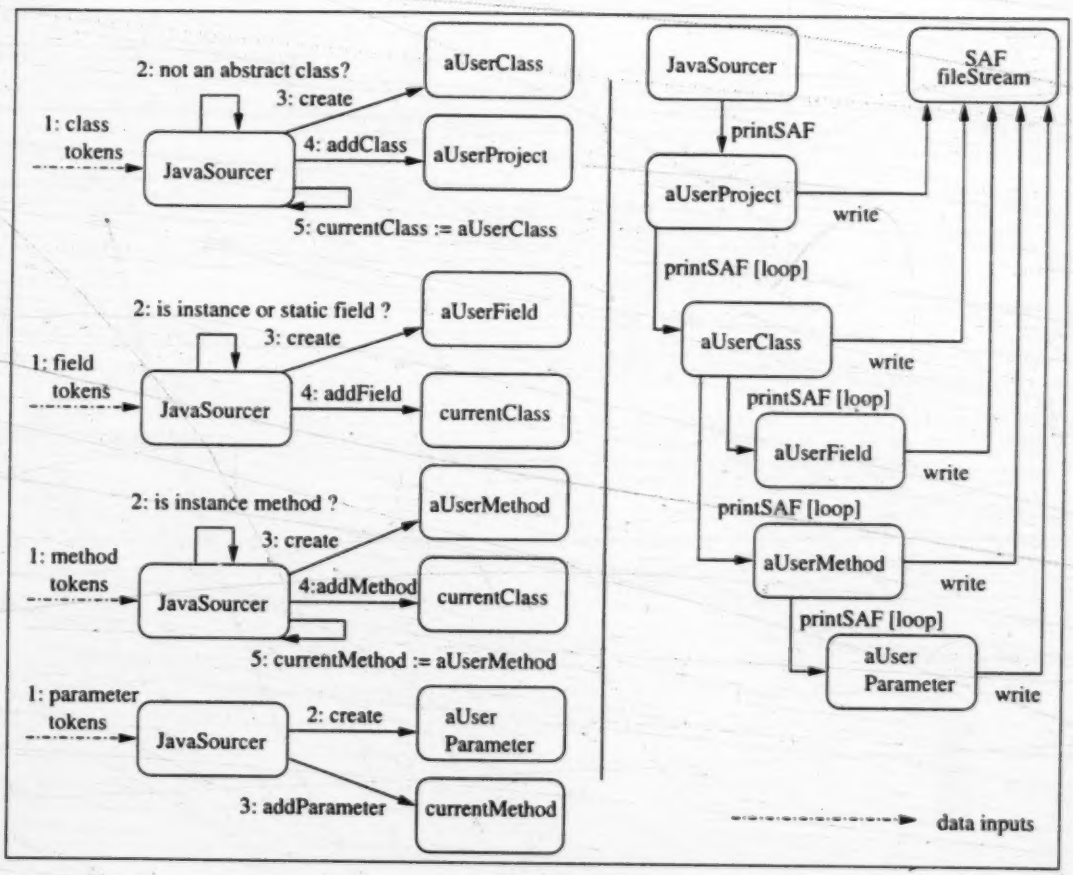


Figure 5.4: Parsing And Extracting Static Artifacts

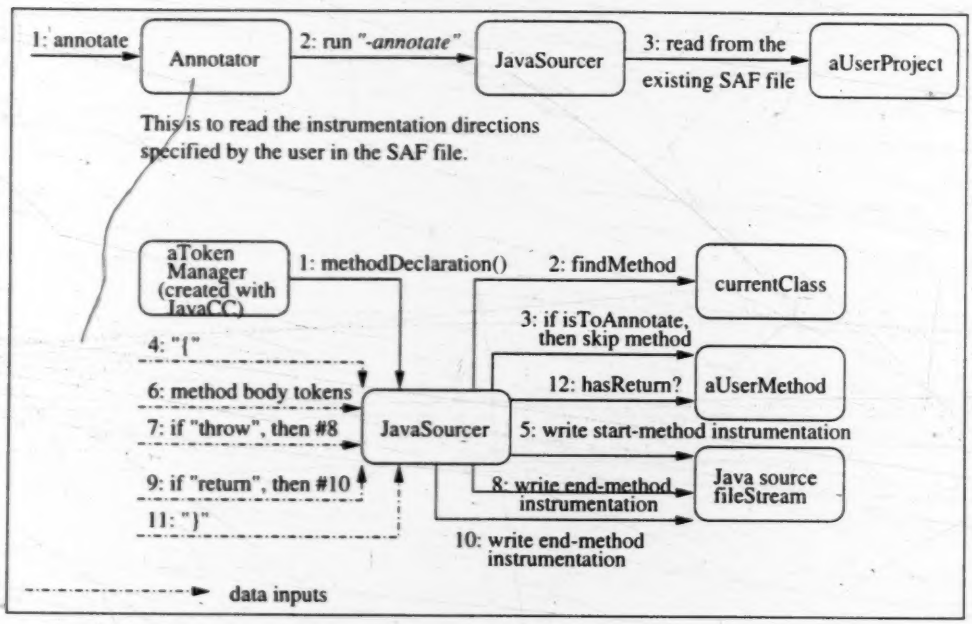


Figure 5.5: Annotating a User's Method

5.4 Implementation

In this section we discuss some of the challenges experienced during the implementation of the subsystem and the solutions that were employed. Examples that show the original code as well as the instrumented version are in Appendices A to C.

5.4.1 Type Table

Revy uses a table to store information about all of the types in the user's source code. It is called a type table rather than a class table because many object-oriented languages are hybrid and have different type/class concepts. For instance, in Smalltalk everything is an object and inherits from the Object class. In C++, there are three categories of types: 1) primitive types like `int` or `float`, 2) structures/classes, and, 3) pointers to structures/classes. Java is somewhere between Smalltalk and C++ with: 1) primitive types, and 2) references to objects which inherit from the class Object.

Revy generalizes the type concept by representing each source program type as

an object, namely `UserType`. Each type object is classified into one of the three categories: Primitive, System or User Types. In addition to the classification, a `UserType` object stores the name of the type and its size in bytes. The size of a type refers to the static size, not the runtime object size. The differences are discussed in Section 5.4.3. All this information is needed to instrument the code as well as to analyze the object granularity properly. For instance, parameter sizes, which is the total number of bytes needed to represent all of the arguments in a message, are used in computing the size of a message.

Instances of Primitive Type represent non-class types like int, char, long, and so on. These types are pre-defined according to the language specification. For example, in Java, there are nine primitive types, including void. Each Primitive Type object stores the actual size of the data items of that type. For example, the size of a char is 2 bytes in Java.

Instances of System Type are types that come with the language library, like Frame and Vector in Java (or, to be exact, any type that is not found in the user source code). Since the source code for a library type is not accessible, we have fixed the byte size of a System Type instance. Note that all instances of library classes are represented by pointers that are 4 bytes in size. Therefore, at first glance, it appears that we should use this size. However, the actual object size is usually more than 4 bytes. We are interested in method granularity for the purpose of remote execution. We cannot actually pass a pointer as a parameter when a method is executed remotely. Instead, we must serialize the object and pass the number of bytes the object actually has. Currently, all system types have been arbitrarily assigned the size of 32 bytes. Some statistical data is necessary to determine the best size to use for library objects, nevertheless, this value serves as a starting point. Note that we do not model each individual System Type separately. Instead, they are all classified under the same umbrella of a unique type model, with the name "`_system_`".

Revy models each User Type[5] object individually. Since we have full access to

[5]For system and user types, the term "type" is being used interchangeably with the term "class".

the source code, every instance of a User Type is given a name and static size. The calculation of the static size of a User Type object is discussed in Section 5.4.3. The static size of a User Type is the lower bound of the size of an instance of that type.

5.4.2 Annotating Classes

We use multiple inheritance [6] to instrument classes. Every annotated user class inherits from the class `_RvObject` in addition to its original superclass(es). Each of the user classes has one additional object ID (OID) field and five additional instance methods. They are `_rvOID`, `_rvClassName()`, `_rvGetOID()`, `_rvSetOID()`, `_rvStaticSize()` and `_rvRuntimeSize()`. The method `_rvSetOID()` is called only once by an instrumented constructor when an object is created. The unique object ID is provided by the `RevyLogger` object.

There is always a tradeoff between using accessor methods or accessing the fields directly. All the above methods except for `_rvRuntimeSize()` can be replaced by direct accesses to the corresponding public instance fields at runtime. However, storing the static size and the class name with each instance is wasteful of space. The annotation therefore inserts additional methods rather than introducing extra fields. Although methods cost code space, we assume users' applications create enough instances to offset extra code space. Nevertheless, the addition of instrumentation code results in the well-known probe effect when the code is executed. That is, the instrumentation takes extra memory and time, and will change the runtime characteristics of the application. The details of the probe effect are discussed in Section 5.5.3.

5.4.3 Object Size

Object size can be queried with the instance method `_rvRuntimeSize()` which computes the number of bytes in the object at runtime. Similarly, the `_rvStaticSize()` method returns the static size of a UserType object as a lower bound of the size of

[6] Although Java does not support true multiple inheritance, its interface concept is sufficient to create the desired effect.

an instance. The reason we have implemented two object size routines is to allow users to fine-tune their instrumentation against the introduced probe effects (refer to Section 5.5.3 for details). Both calculations use a similar algorithm. The only difference is that the static size algorithm generates the compile-time size and the runtime size method calculates the size by executing the algorithm at runtime.

Figure 5.6 shows how the static size of an object is calculated. The algorithm first collects the set of all instance and static[7] fields defined natively in the class, and all instance and static fields inherited from the superclasses. (That is the statement "`this.getAllFields()`".) It is important to identify the static fields as well as the instance fields since when an object is passed as a parameter to a remote message, the class information must be sent to the remote site along with the instance information. By collecting the fields into a set, any duplicate fields are ignored. For example, there will be only one count of the field `id` of `Student` if it derives from the class `Person`, providing that both classes have defined their own field `id`.

Once all the fields are identified, PAS recursively queries the object referenced by each field, asking for its runtime size. (In the case of static size, PAS recursively queries the `UserType` object of each field, i.e. the statement in the while-loop: "`f.getType()._rvStaticSize(visitedSet)`".) Note that this recursive method is called with a parameter – a visited-object set (or a visited-class set, in the case of static size) – so that cycles can be identified during this object traversal. (Refer to the base case statement "`if(visitedSet.contains(this)) ... return ...`" in the figure.)

5.4.4 Annotating Methods

Revy allows users to choose which methods to instrument. From each instrumented method call, we must identify five pieces of information: the method, the receiver object, the sender object, the duration of the call, and the total size of the parameters.

[7] In Java, final fields are not considered since we assume the compiler will optimize the code such that an instance will not carry any final fields with itself.

```
/* UserType method:  calculate lower bound size of an instance */
public int _rvStaticSize(Set visitedSet)
{
  /* base case */
  if(visitedSet.contains(this)) { return(POINTER_SIZE) ; }
  visitedSet.add(this) ;
  /* first collects all native and superclass' fields */
  e = this.getAllFields() ;
  /* then recursive call on the type of each field */
  while(e.hasMoreElements())
  {
    f = (UserField) e.nextElement() ;
    statSze = statSze + f.getType()._rvStaticSize(visitedSet) ;
  }
  return(statSze) ;
}
```

Figure 5.6: An Example of Finding Static Object Size

The receiver object in a method is easy to identify since all object-oriented languages provide a pseudo variable that is bound to it (the variable is called `this` in both Java and C++). Identifying the other information is not quite so straightforward.

The method name and receiver class are sufficient to uniquely identify methods in some languages like Smalltalk, but not enough in languages which support static multi-methods. Static multi-methods exist in some languages that support static typing. In those languages, methods can have the same name but have different parameter types. Therefore, we generate a signature for each method based on its name, static receiver type and static parameter types. We do not include the static return type since method dispatch ignores return types.

We compute the duration of a message by inserting start-method and end-method instrumentation code in the method. These two instrumentations will record the start and end times of the message, and hence the duration of the message can be determined by the time difference. For a start-method instrumentation, we can simply

insert the code at the beginning of the method body. However, the insertion of an end-method is not as easy as it seems. A method may have no return, one return, conditional return or exceptional return statements. Our Sourcer identifies all potential return points and inserts an end-method log before each of them.

In order for Revy to build the object call graph, we need to identify the sender of a message. In Smalltalk, recording the sender object for each method invocation is easy since there is a built-in method `#sender` which returns the caller of the active method. In Java and C++, no such method is available. However, given two sequential start-method log entries, we know that the sender of the second method is the receiver of the first one. This simple observation is modified slightly in the case of Java where the language supports multiple threads. In this case, we need another piece of information, the thread ID, for each method. The details of how traces are translated will be discussed in Section 6.4.2.

5.4.5 Message Size

The size of a message is the sum of the sizes of all of its parameters, since each of these parameters must be serialized into a byte stream when a method is invoked remotely. Message size is computed at runtime. Each parameter is asked for its runtime type, and the size of the parameter is determined by the classification of its type. The sizes of Primitive and System Types are fixed and pre-determined as discussed in Section 5.4.1. The size of objects with UserType has been discussed in Section 5.4.3. Since this calculation is performed at runtime, it can potentially create a major probe effect, especially if a collection of objects is passed as a parameter. To reduce the probe effect, we provide the user with an alternative solution. Instead of computing the actual size of the collection of objects, we can sample the collection to compute an estimated size. The sampling algorithm asks the size of the first element of the collection, then multiplies that size by the length of the collection. The size of an array can be correctly determined in Java by accessing its public field `length`. However, this is not possible in C++ and hence we assume all arrays are of fixed

```
method(int a, char b[], Frame c, Person d, Student e[][])

int _approxSize =
  4 +               /* a:  int is primitive, so fixed */
  2 * b.length +    /* b:  char is primitive, so fixed */
                    /* b:  array, so multiplied by length */
  _DFLT_SZE +       /* c:  frame is system type, so fixed */
  ((d == null)?  _NULL_SZE:d._rvStaticSize()) +
                    /* d:  Person is user type, so query */
                    /* d:  make sure not null first */
  ((e[0][0] == null)?  _DFLT_SZE:e[0][0]._rvStaticSize())
    * e.length[0] * e.length ;
                    /* e:  array & user type, so sampling */
                    /* e:  make sure 1st element not null */
                    /* e:  then multiplied by the length */
```

Figure 5.7: Finding Method Size by Sampling

length[8]. Figures 5.7 and 5.8 show how a method size is calculated approximately and exactly (assuming `Frame` is a System Type, while `Person` and `Student` are User Types). Note that both solutions – the accurate but expensive, and the fast but approximate – are available to the user for fine-tuning the instrumented application.

5.5 Problems and Concerns

5.5.1 Problems with Tools

Both Sage++ and JavaCC have problems that needed to be worked around. This section describes these problems in turn.

Sage++

Sage++ is a powerful source-to-source translator tool. It was used for our C++Sourcer. Sage++ provides a fairly complete library of functions, and developers can use it to parse files written in C, C++ or Fortran. The main problem with their library is its

[8]The default collection size is 8, and it is a user-configurable parameter.

```
method(int a, char b[], Frame c, Person d, Student e[][])

int _exactSize =
  4 +              /* a:  int is primitive , so fixed */
  2 * b.length +   /* b:  char is primitive , so fixed */
                   /* b:  array, so multiply by length */
  _DFLT_SZE +      /* c:  frame is system type, so fixed */
  ((d == null)?  _NULL_SZE:d._rvRuntimeSize()) ;
                   /* d:  Person is user type, so query */
                   /* d:  make sure not null first */
  for(i=0 ; i < e.length ; i++)
    for(j=0 ; j < e.length[i] ; j++)
      _exactSize = _exactSize +
        ((e[i][j] == null)?  _NULL_SZE:e[i][j]._rvRuntimeSize()) ;
                   /* e:  array & user type,loop inside */
                   /* e:  Student is user type, so query */
                   /* e:  make sure not null first */
```

Figure 5.8: Finding the Exact Method Size

lack of object-orientation. Sage was first written in C, and the developers changed to C++ by wrapping structures with classes. Hence, Sage++ is not really object-oriented. Another limitation of Sage++ is that it fails to support many features in C/C++. For example, the class **iostream** is not supported, and therefore, we have to assume users do not write any **cout** or **cin** statements. This is a serious limitation in practice. Similarly, Sage++ cannot resolve **#include** statements. Users have to use a preprocessor like "CC -E" to merge the include files before feeding the source into Sage++. However, the processed source is still not clear enough for Sage++ as it usually contains **#pragma** comments that have to removed. In essence, Sage++ is powerful yet it does not work well with code written by a typical C++ programmer.

JavaCC

JavaCC, which is essentially YACC in Java, is a basic parser generator tool. Our Java-Sourcer was written with JavaCC. Being supplied with a language grammar, JavaCC can parse a file written in that language. Unfortunately, it provides developers with

just a few library functions, and forces users to build/rebuild a lot of routines. There are pros and cons to YACC in Java. Programmers can write embedded Java code in YACC-style JavaCC programs. Despite the fact that it is written in an OO language, the design of JavaCC does not seem to be object-oriented. Users are forced to write static methods which, in fact, are functions in non-OO languages. As Java is 20 times slower than C [FL96], so are JavaCC programs since they are written in Java. Nevertheless, JavaCC is an appropriate choice for two reasons. The main reason is that JavaCC, written in Java, supports Unicode escapes. Therefore, we are guaranteed to parse the user's sources as well as to generate new sources correctly. Secondly, our primary target language is Java, and JavaCC allows us to modify sources with embedded Java code. In essence, JavaCC is powerful enough to be used in this project.

5.5.2 Language Problems

Different languages have different strengths and weaknesses. From the perspective of Revy requirements, we will discuss them in the following categories: classes, methods and parameters, and exceptions.

Classes

Some languages include a class reference or a class name in each of the instances created, like Smalltalk and Java, while some do not, like C++. If this is supported by the language itself, we do not have to attach a field/method with every object/class, and thus the object overhead is not increased. However, Java gives us more than we need – it gives the class name together with the package name. For example, the `aVector.getClass().getName()` message returns not "`Vector`" but "`java.util.Vector`". We therefore keep our own class name copy for each class.

True multiple inheritance gives developers higher flexibility. However, it also makes the analysis of a user's code more complex. C++ supports true multiple inheritance while Java simulates it, and Smalltalk supports none. From the Revy system point of view, the inheritance model in Java is sufficient to create the desired

effect. For example, an instance of the `Thread` class has two "hats" (two types), namely, `Thread` and `Runnable`, because the definition of the class `Thread` implements the interface `Runnable`. We call this type of inheritance model "multiple-typing", as it makes all instances belong to exactly one class and can be tagged with more than one type. Multiple typing is just what is needed in Revy. All the Revy log routines are invoked with the object `this` in the user's code. However, it does not make sense to define different log routines for each possible type of `this`. Hence, we have made all user's classes implement the interface `_RvObject`, and we have defined only one single log routine which takes the `this` object as of the type `_RvObject`.

Methods and Parameters

The major problems with a method are its signature and parameters. Since both Java and C++ support static multi-methods, we have to uniquely identify a method by generating a method signature. Fortunately, the generation of a method signature is not expensive as it is a one-time process during the parsing phase. However, finding the size of the passed parameters can be costly and it requires runtime calculation. Furthermore, while we are able to calculate the exact parameter size in Java, it is impossible in C++. The key difference is that in Java we can ask a parameter for its class and the length, if it is an array. In C++, parameters can be pointers, structures or even pass-by-reference types. We have no way to tell whether a pointer actually represents an array or not. If so, what is the length of the array, and does it contain pointers to other arrays? These questions are not answerable in C++.

Exceptions

One of the concerns about any programming language is its support for exceptions. Exception handling mechanisms make applications more robust, but the control flow can be hard to manage. Explicit exceptions in a method are similar to return statements, and they need to be identified in Revy. However, implicit exceptions may not be easy to handle. For example, Java has a well-defined list of implicit exceptions like `NullPointerException` and `IllegalArgumentException`. Since Java is a statically

typed language, the compiler is able to filter out some implicit exceptions such as illegal-type or no-such-method exceptions. Users can also raise (or "throw" in Java terminology) their own exceptions provided they have declared them in the method header. Therefore, identifying implicit exceptions in Java is not difficult. In contrast, Smalltalk has a large set of implicit exceptions, and no static typing. Hence, identifying and trapping exceptions is not as easy in Smalltalk.

5.5.3 Probe Effects

Probe effects will occur in the Revy system due to the code inserted to collect runtime traces. There are two sources for probe effects: object and method overheads. Object overhead consists of the additional methods and data fields attached to each object. This increases the compiled code size and the runtime memory requirements. Our implementation adds one field and five methods to each instrumented class. In fact, the two message size computation methods can be omitted if the class to be instrumented inherits from another instrumented class, provided that it does not define additional native fields. Such cases are quite rare, and so we include the methods in any case.

Method overhead mainly originates from three sources: computation of parameter size, generation of a log string, and I/O access. To amortize the I/O cost, we use buffers, i.e. the classes `StringBuffer` and `BufferedOutputStream` in Java. During the generation of a log string, several costs accrue: system clock access, thread access, and object and method details inquiries. These costs are almost un-avoidable except to shorten the log strings. The computation of the parameter size involves querying the size of each parameter dynamically. We also provide a tradeoff solution as mentioned in Section 5.4.5: estimating the size of an array by sampling. Such a tradeoff has justifications because a (potential) parallel application usually uses large amount of numeric data which are passed back and forth as large arrays. Numeric data is fixed in size, and we can always ask for the dimension and length of an array in Java.

To reduce these two probe effects, we can measure the compiled code size, runtime memory requirements and speed of the instrumented program, and compare these aspects against those of the un-instrumented program. Users can always switch from an exact but expensive instrumentation to a fast but approximate instrumentation. The exact instrumentation, which can be selected by passing the "-annotateBetter" flag on the command line, has calls to the `_rvRuntimeSize()` method; while the fast one (that is, the "-annotateFaster" flag), has calls to the method `_rvStaticSize()`.

5.6 Further Enhancements

The Parsing and Annotation Subsystem can be enhanced in several ways. We can be more accurate in the annotation of code when dealing with the language specific issues, such as handling exceptions, and we can also be more exact in logging details, such as method signatures.

Catching Exceptions. Currently, Revy does not handle the implicit exceptions of standard Java and the user's explicit exceptions. For instance, `NullPointerException`, an implicit exception which is usually due to a runtime error, is not caught by Revy and will force the user's program as well as `RevyLogger` to terminate. Explicit exceptions which are declared in the method header are not handled by `JavaSourcer` either. Explicit exceptions will not cause the user's program or `RevyLogger` to fail, but the current active method will abort. As a result, we will fail to log the end of the method. For both situations we could implement a catch-all solution. For each single method, including the `main()` function, we could catch all Java implicit exceptions and all the user's explicit exceptions defined in the method header. After the catch, we could re-raise whatever exceptions we have caught. Figure 5.9 shows the instrumented code. We can see that the resulting code looks very complex, and is hard to read and understand. For C++, we do not handle any exceptions.

Generating Signatures. The current implementation of PAS identifies a method by its method name, class and parameters' types, but groups all system types together.

```
/* Original Code */
public void method() throws FileNotFoundException
{
  f = new FileInputStream("foo.java") ;
}
```

```
/* Instrumented Code */
public void method() throws FileNotFoundException
{
  try
  {
    try
    {
      f = new FileInputStream("foo.java") ;
    }
    catch(FileNotFoundException e1) {_rvLogEnd(...)  ; throw e1 ;}
  /* all user explicit exceptions go here */
  }
  catch(IllegalPointerException eN) {_rvLogEnd(...)  ; throw eN ;}
  /* all Java implicit exceptions go here */
}
```

Figure 5.9: Catching All Exceptions

However, such simplification may confuse the logger. Figure 5.10 shows two methods (the second and third methods) which have the same method name and same number of arguments, but use different system types. These two methods end up with the same method signature, because PAS does not distinguish between individual system types and it collectively names them "`_system_`". A solution is to assign a separate name for each system type.

5.7 Summary

In this chapter, we discussed one of the key components of the Revy system – the Parsing and Annotation Subsystem. We analyzed its requirements, presented the architecture and provided a design solution. Moreover, we discussed the challenges,

```
Format of a method signature:
  !type^arrayDimension!type^arrayDimension!...

1.public int hire(int i[], Employee e, Vector v)
  signature:  !_system_^1!Employee^0!_system_^0

2.public int hire(int i[], Student s, Vector v)
  signature:  !_system_^1!Student^0!_system_^0

3.public int hire(int i[], Student s, Hashtable h)
  signature:  !_system_^1!Student^0!_system_^0
```

Figure 5.10: Method Signatures

problems and concerns we encountered during the development phase. Finally, we suggested a few future enhancements for PAS. PAS was implemented for two different object-oriented languages: C++ and Java. Although, PAS has not been implemented for Smalltalk, its object model was considered. Therefore we were able to generalize most concepts of typical OO languages. Being provided with the well-defined interface files, PAS can be a stand-alone application. We believe that PAS can be extended so that it will become a more powerful and generic source-to-source translation tool for OO languages.

Chapter 6

Runtime Modeling Subsystem

While PAS instruments a user's object-oriented program, the Runtime Modeling Subsystem (RMS) of Revy traces and interprets the dynamic behaviour of the program for the user to visualize using VIS. This chapter lists the requirements of RMS, and presents its architecture, design and implementation. In addition, we discuss some of the outstanding issues and suggest a few further enhancements of RMS.

6.1 Requirements

The goal of the Runtime Modeling Subsystem is to collect the runtime statistics of an OO program and construct the object communication patterns (an object call graph) for the Visualization and Interaction Subsystem to present to the user. To achieve this, RMS is required to:

- log the creations of instances and the invocations of methods instrumented by the Parsing and Annotation Subsystem to a trace file while the user's program is running,
- translate the trace file to a logical object call graph for VIS, and
- process, calculate and sort the runtime statistical data of the instances and messages for VIS to display.

The modular design of Revy implies a requirement that the subsystem should be able to start and be tested even without a GUI. In addition, the Revy system imposes

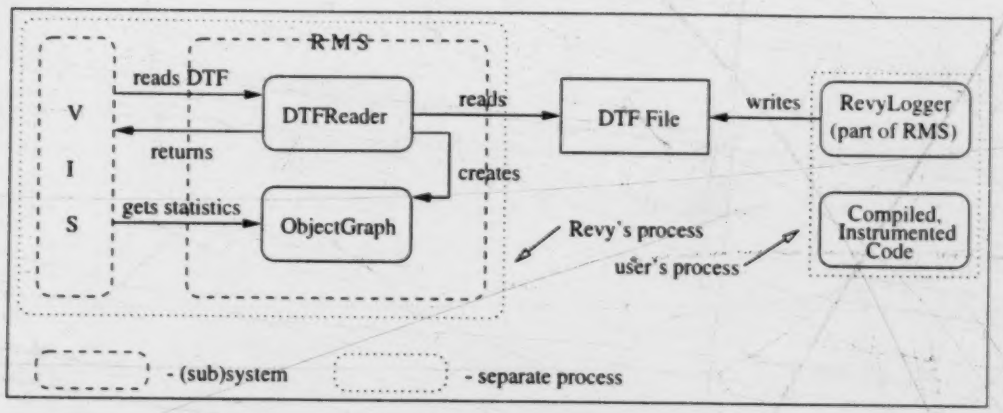


Figure 6.1: RMS Architecture

two interface requirements: (1) a physical file, Dynamic Trace File (DTF) in which the runtime traces are saved, and (2) an `ObjectGraph` instance which is a logical representation of the DTF file. These are the required interface between RMS and the rest of the system.

6.2 Architecture

The Runtime Modeling Subsystem has two components: the majority of RMS belongs to the Revy process, and the rest is part of the user's application process, as shown in Figure 6.1. The `RevyLogger`[1] object is responsible for writing the runtime traces to a DTF file. `DTFReader` reads the DTF file and creates an `ObjectGraph` instance. `ObjectGraph` stores the object communication pattern, and computes and processes the runtime statistics used by the Visualization and Interaction Subsystem. Note that there exists a pipeline pattern in the RMS architecture diagram: the `RevyLogger` object writes to a DTF file while the `DTFReader` object reads from the file. Such a pattern is ideal for asynchronous processing. Nevertheless, the current implementation of the Runtime Modeling Subsystem is a post-mortem analyzer.

[1]The actual class name of `RevyLogger` is `_RvLogger`.

Instance Creation Log

```
<Crt cls=CLASS oid=OBJ_ID sze=INSTANCE_STATIC_SIZE tid=THREAD_ID>
e.g.:  <Crt  cls=Account  oid=6  sze=16  tid=0>
```

Instance Destruction Log

```
<Dst cls=CLASS oid=OBJ_ID tid=THREAD_ID>
e.g.:  <Dst  cls=Account  oid=6  tid=0>
```

Method Invocation Log

```
<Msg cls=CLASS oid=OBJ_ID mth=METHOD sgn=METHOD_SIGN sze=METHOD_SIZE tid=THREAD_ID t=TIME_STAMP>
e.g.:  <Msg  cls=Account  oid=6  mth=transfer  sgn=!double^0!Account^0!  sze=24  tid=0  t=293>
```

Method Return Log

```
</Msg cls=CLASS oid=OBJECT_ID mth=METHOD sgn=METHOD_SIGNATURE tid=THREAD_ID t=TIME_STAMP>
e.g.:  </Msg  cls=Account  oid=6  mth=withdraw  sgn=!double^0!  tid=0  t=295>
```

Figure 6.2: Format of the Dynamic Trace File

6.3 Design

In this section, the design of the Dynamic Trace File (DTF) is presented, and the software design of the classes of RMS and their interaction are also discussed.

6.3.1 Dynamic Trace File

The required physical interface between VIS and RMS is a Dynamic Trace File (DTF), which can have four types of entries: (1) instance creation, (2) instance destruction[2], (3) method invocation, and (4) method return. Each entry in a DTF file occupies one line. The format of a DTF file is shown in Figure 6.2. Note that a DTF file is stored as text format with HTML conventions. The embedded HTML tags allow a DTF file to be easily viewed with a modified Web browser. The text format allows users to read and understand the context. Hence, users can inspect the log details in the trace file manually. A sample of a DTF file is given in Appendix F.

[2]Only exists for languages that have explicit destructors, like C++.

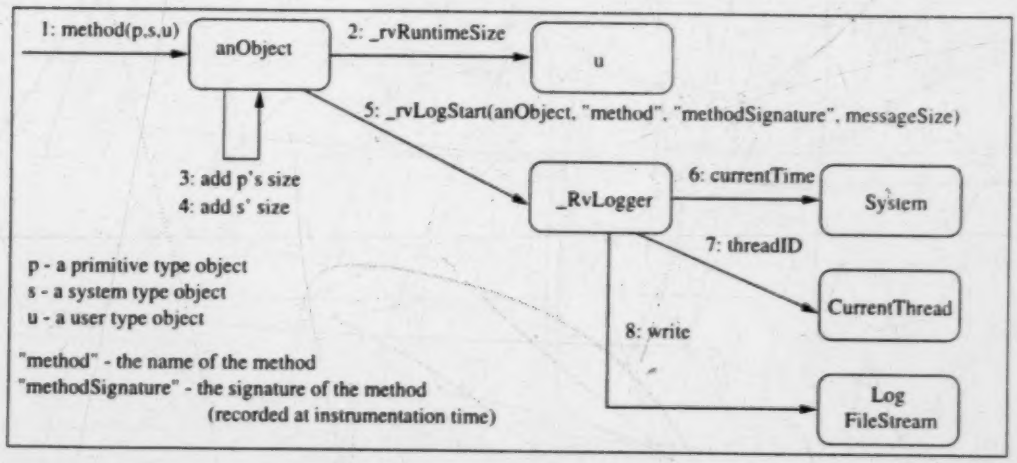


Figure 6.3: Logging Runtime Traces

6.3.2 Class, Responsibility and Collaborators

The classes of RMS can be classified into two categories: supporting and major classes. The supporting classes, `InstanceModel` and `MessageModel`, are responsible for modeling the instances and messages, respectively, as recorded in the trace file. These classes are used in conjunction with the supporting classes of the Parsing and Annotation Subsystem (PAS): the `UserClass` and `UserMethod` classes. Appendix K presents the RMS supporting classes. The major classes of RMS are the `RevyLogger`, `DTFReader` and `ObjectGraph` classes. These classes are major in a sense that they perform most of the processing tasks in the subsystem. Appendix L shows the major classes of RMS.

6.3.3 Collaboration Diagrams

The collaboration diagrams of RMS are presented according to the three requirements stated in Section 6.1. Figure 6.3 shows how the `RevyLogger` object is invoked to record the runtime behaviour of a user's program. The tasks of the `DTFReader` object are presented in Figure 6.4. Figure 6.5 demonstrates how the `ObjectGraph` instance is queried for the runtime statistics.

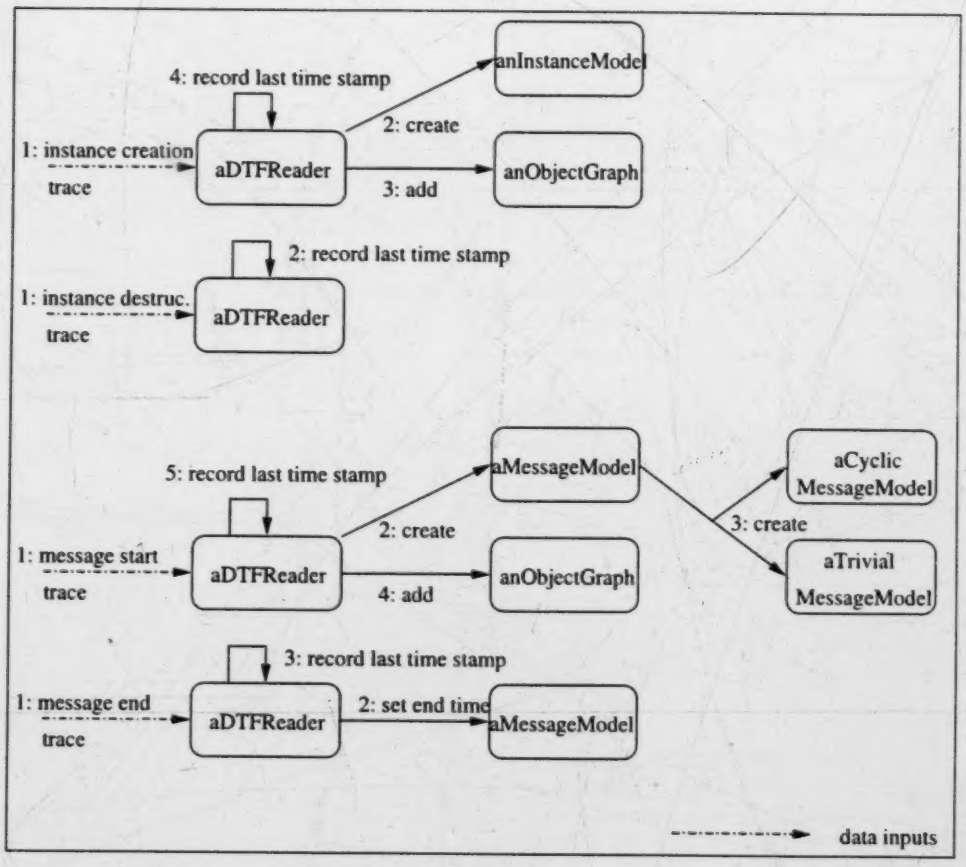


Figure 6.4: Translating Traces

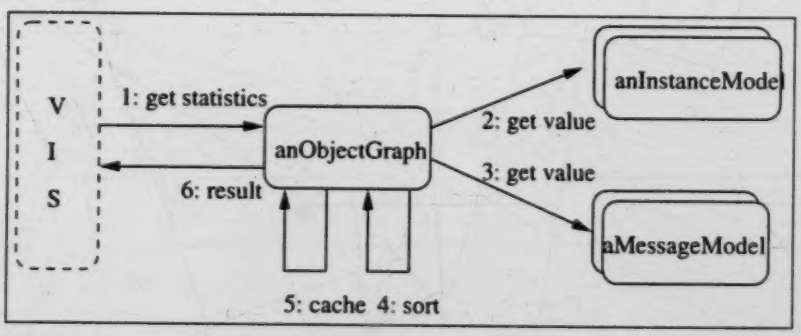


Figure 6.5: Querying Statistics

6.4 Implementation

In this section, we discuss the challenges that arose during the implementation of the three tasks of RMS: logging runtime traces, translating the trace file to a call graph, and the collection of runtime statistics.

6.4.1 Logging Runtime Traces

The logging responsibility of RMS is to record the invocation of each instrumented method, whether the method is a constructor or a regular method. The logging routines are attached to the user's code, therefore the implementation depends on the user's programming language. The `RevyLogger` class implements the logging routines. A natural OO design is to have all user class objects derive their logging behaviour from `RevyLogger`, since each instance is responsible for logging its own messages. However, some user's classes already inherit from other classes. To preserve the semantics of the user's program, the natural OO design can only be implemented if the programming language supports multiple inheritance. However, multiple inheritance does not exist in all OO languages. Although C++ supports multiple inheritance, Java does not. Therefore, a generic `RevyLogger` is implemented as a stand-alone class. None of the user's classes derive from `RevyLogger`, and therefore none of the user's objects inherit the logging methods. The user's objects can access the logging routines directly since the routines are implemented as static functions. Static functions are faster than instance methods and therefore the efficiency of the logging process is increased.

6.4.2 Translating Traces

The translation process takes a DTF file as input, parses it and creates an `ObjectGraph` instance. Note that each entry in the DTF file is on a single line and corresponds to one method call. A DTF file can therefore be perceived as a message stack. The receiver of the current message is recorded in the object ID attribute of the line. The sender of the current message is identified as the receiver of the outstanding message,

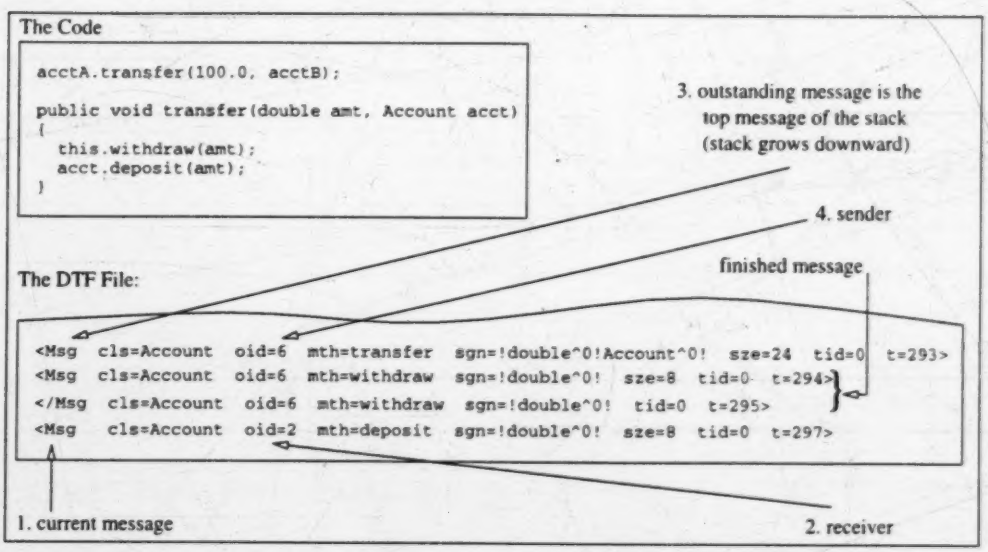


Figure 6.6: Identifying Message Sender

which is the top message on the message stack. A snapshot of a DTF file together with a brief explanation of the translation of the algorithm is presented in Figure 6.6. This algorithm was modified slightly in the case of Java, where multiple threads are supported. In this case, the message log entries from different threads may be interleaved. However, logically there is one message stack for each thread. Therefore, we store the thread ID for each thread, and the rest of the algorithm is same.

To resolve the timings, we record the time stamp of the last log entry for each thread, and label it as the "last time stamp". The net processing time of a message is calculated by taking the last time stamp, subtracting it from the time stamp of the current message, and accumulating the result as the net processing time of the outstanding message. The execution time of an instance is the sum of the net processing time of all its messages. To find out the aggregate processing time of a message, we simply record the time stamp of its start-message log as the message start time, and the time stamp of its end-message log as message end time. The aggregate time is then the difference between these two time stamps.

6.4.3 Call Graph and Statistics

After the `DTFReader` has built an `ObjectGraph` instance and calculated individual objects' statistics, post-parsing routines can be executed. For example, the total application execution time, the activity level of each message and instance, and the adjacencies of instances can be computed. In the future, we can implement other post-parsing routines such as the groupings of instances and messages.

The `ObjectGraph` instance is responsible for returning the results of the queries from the Visualization and Interaction Subsystem. The current implementation includes: instance size, instance execution times, instance average execution times, message size, message net and total processing times, and message average processing times. In addition, other information is available, such as object IDs, message orders, adjacencies of instances, and so on.

Our implementation includes a generic sorting routine, which sorts any one of the above statistics before VIS presents it to the user. During the testing phase, we noticed that users may sometimes re-query the system for the same statistics. Hence, an `ObjectGraph` keeps a cached copy of the sort results to avoid repeated sortings (see Figure 6.5).

6.5 Concerns and Further Enhancements

In this section, we discuss the problems and outstanding issues we have encountered during the development of RMS. The problems are described according to the functionality of each object.

6.5.1 `RevyLogger`

The main concern about the logging facility is the probe effect. Recall that probe effects can occur when the code inserted to collect runtime traces is executed. Several techniques are employed to reduce the effect, as described in Section 5.5.3. Despite

these techniques, the size of the probe effect for Java as reported in Chapter 7 is roughly 2 to 3 milliseconds per logging routine call. To further reduce this overhead, the logger can be modified to write binary data instead of text data. That is, in the case of Java, it can use the class `ObjectOutputStream` for logging. This technique will save time in two ways: (1) no translation time of objects to text strings, and (2) shorter I/O time due to the fact that the size of the binary representation of an object is smaller than its textual representation. This will also reduce the execution time of the `DTFReader` object as well. The disadvantage of this technique is that the user will lose the ability to easily read the trace files.

Another concern is the robustness of the logger object. Robustness usually conflicts with efficiency. For example, `RevyLogger` uses a buffered I/O stream of size 32 kilobytes. This large size was chosen because testing revealed that a trace file could easily have a size over 10 or even 100 kilobytes. However, this simple buffering scheme is subject to data loss. For instance, as pointed out in Section 5.6, an uncaught exception can cause the user's program, as well as the logger object, to terminate. As a result of such an unexpected termination, some of the trace data in the logger's buffer is lost. There is a tradeoff: either increase the robustness by using no buffer (or a smaller buffer), or increase the efficiency by using a large buffer. This project assumes that the user provides a sound program with valid inputs, and therefore the program should execute until it terminates gracefully, and the trace file should be complete as well. Hence, a larger buffer for `RevyLogger` is used to increase efficiency, but with a chance of loss if abnormal termination occurs.

6.5.2 DTFReader

The current implementation of `DTFReader` is stable. The only concern is its efficiency, as it usually handles large DTF files. There are various areas in which the `DTFReader` can be further enhanced. First, as will be pointed out in Section 7.5.2, the `DTFReader` can be optimized by using fewer classes with built-in synchronized methods. For example, arrays should be used instead of the `Vector` and `Hashtable` classes, provided the size of the collection is known beforehand. Second, the `DTFReader` object could

be parallelized to improve the performance. Third, while the current implementation is a post-mortem reader, the next implementation should be asynchronous with the writing process of `RevyLogger`. In general, the `DTFReader`'s code is complete and correct in the current implementation, and the next evolution is to fine-tune its performance so that users will find it more responsive.

6.5.3 `ObjectGraph`

`ObjectGraph` has dual responsibilities: to store the call graph details, and to compute the runtime statistics. The storage of the graph details is trivial and its code is mature. The statistical calculation is the part that could be enhanced. There exist other opportunities for gathering additional data, such as communication and processing costs. However, we need the user's aid for these types of calculations. For instance, we could make use of the number of machines, communication cost per byte between two machines, relative processing costs per unit of time, and so on. This part of Revy would require future research efforts. Nevertheless, the result of our current research is sufficient for multiprocessor machines with shared memory (SMP machines), because the communication costs between two CPUs is negligible, and the processing power of all CPUs are the same.

Other future research could focus on load balancing and graph partitioning algorithms, such as those discussed in [CT92]: Bin Packing, Randomization, Pressure Model and the Manager-Worker scheme, and those discussed in [SB96]: Heaviest-Edge-First (HEF), Minimal Communication (MC), Kernighan-Lin (KL). By using these algorithms, Revy would become a more complete parallelization advisor. Finally, if the future `ObjectGraph` is enriched in these ways, its computation responsibility should be separated from its call graph storage responsibility by creating a new class called `GraphAnalyzer`.

6.6 Summary

This chapter discusses the Runtime Modeling Subsystem (RMS) of Revy. The three requirements of the subsystem: logging traces, parsing traces, and computing runtime statistics were presented. In addition, its architecture, design and implementation details were provided. Moreover, we discussed the problems and concerns encountered during the development phase, and suggested some further enhancements for RMS. Although, there exist many opportunities for improvement, RMS provides an object call graph as a basic framework for the application of parallelization algorithms. Further enhancements would transform Revy into a better program parallelization advising tool.

Chapter 7

Experimental Results

The Revy System is a parallelization advisor which provides runtime statistics to users. The usefulness of the statistics cannot be justified unless we perform experiments on some applications using Revy. This chapter outlines the results from the testing of three applications. Other issues and problems that arose during the course of the experiments are also discussed here. Although these issues are not related to the main research goals, they are noteworthy, and further demonstrate some precautions in implementing and parallelizing an Object-Oriented system. The experimental results show that the stated requirements of Revy have been met, and justify that users can have a better understanding of any parallelization bottleneck in their programs.

7.1 Test Bed

The testing employed three applications, namely the DTF file reader component of Revy, an HTML file parser and a zip program. All of the applications were written in Java. The DTF file reader is the one discussed in Chapter 6. Hence, we tested Revy on Revy itself. The HTML file parser and the zip program were obtained from the Internet, and they can be found at [KO96] and [LE96], respectively. On all the applications, we applied Revy to obtain runtime statistics, parallelized the code manually based on the hints gained from examining the Revy output, and compared the sequential and the parallel performance. In addition, we also measured the probe effects due to instrumentation.

The testing used two Java version 1.1.3 Virtual Machines. One is the regular version with the Java Green Threads (JGT) package, and the other VM is packaged with a beta release of Java Native Threads (JNT). The JGT package implements a coarse time-slicing technique to execute many VM threads which map to only one kernel thread on a single processor. The JNT package uses the many-to-many multi-threading model [SS97] on a multi-processor machine. JNT allows a program to have as many threads as appropriate, and each thread is mapped to exactly one kernel thread. For all experiments, we used Java's default start-up parameters, such as 16 megabyte maximum heap size, 1 MB initial heap size, and so on. The experiments were performed on a Sun UltraSparc station with two 200 MHz SuperSparc processors, 128 MB of main memory, and running Solaris 2.5.1. The testings were timed using the UNIX C-Shell built-in command "`time`". For each application, we compared its execution time in three settings: (1) the sequential version running with JGT, (2) a parallel version of the same program with JGT, and (3) the same parallel version running with JNT.

7.2 DTFReader

7.2.1 Program Description

DTFReader is an object-oriented program which reads in a DTF file and constructs an object graph in memory without the GUI. It consists of 16 classes and 174 public instance methods. Given a small DTF file, it can generate a huge runtime trace file if full instrumentation is requested. The following sections discuss the analysis and experimental results of the DTFReader.

7.2.2 Runtime Heuristics

The OO nature of the DTFReader causes the program to have a very fine granularity. With a 1.7 KB input file, the DTFReader produces a 181 KB trace file. The original object call graph generated from the trace file was too cluttered, so we had to re-arrange the objects, and apply the filtering function to hide all simple in-

stances and "get"/"set" messages. Figure 7.1 shows a screen shot of the resulting call graph. The graph, which originally consisted of 131 instances and 1028 messages before filtering, had 19 instances and 51 messages after filtering. We highlighted a few instances and messages which were the most active in the system, and thus were the potential objects to be parallelized. We then inspected their actual active times. Figure 7.2 shows the active times of all instances and the aggregate (total) active times of all messages. The runtime statistics show that the most active instance was `DTFReader:36`[1], with an active time of over 7000 milliseconds. The second most active instance, `ObjectGraph:33`, had an active time of 528 milliseconds. For this example, it would not be beneficial to create `ObjectGraph` on a separate thread, since it was active only for a few hundred milliseconds and constituted only 4.6% of the total active time of the application. We estimate that an object has to be active for over 1000 milliseconds in order to justify the overhead of spawning a new thread.

The messages that executed for about 1000 milliseconds or more are `5:run()`[2], `6:read()`, `8:readLines()`, `870:doStatistics()` and `927:doMessageStats()` (see Figure 7.2). They were the candidate messages to be executed remotely. However, all of them were circular messages of `DTFReader`. That is, they were sent and received by the same `DTFReader` instance. If a `DTFReader` instance is spawned remotely, all its circular messages will execute on the remote thread as well. Therefore, either the messages or the instance could be executed remotely, but not both. At this point, we have examined all instances and messages that executed over 1000 milliseconds, and `DTFReader` is the final candidate for parallelization. The test results of distributing the instances of `DTFReader` are presented in the next section.

7.2.3 Parallelization Results

The application was parallelized by making the class `DTFReader` a subclass of `Thread`. The test program was run for three different times with an input of: one, two and four 180-kilobyte DTF files. The program created one, two and four threads, respectively.

[1]The number following the class name of an object is the object ID.
[2]The number preceding the method name of a message is the message ID.

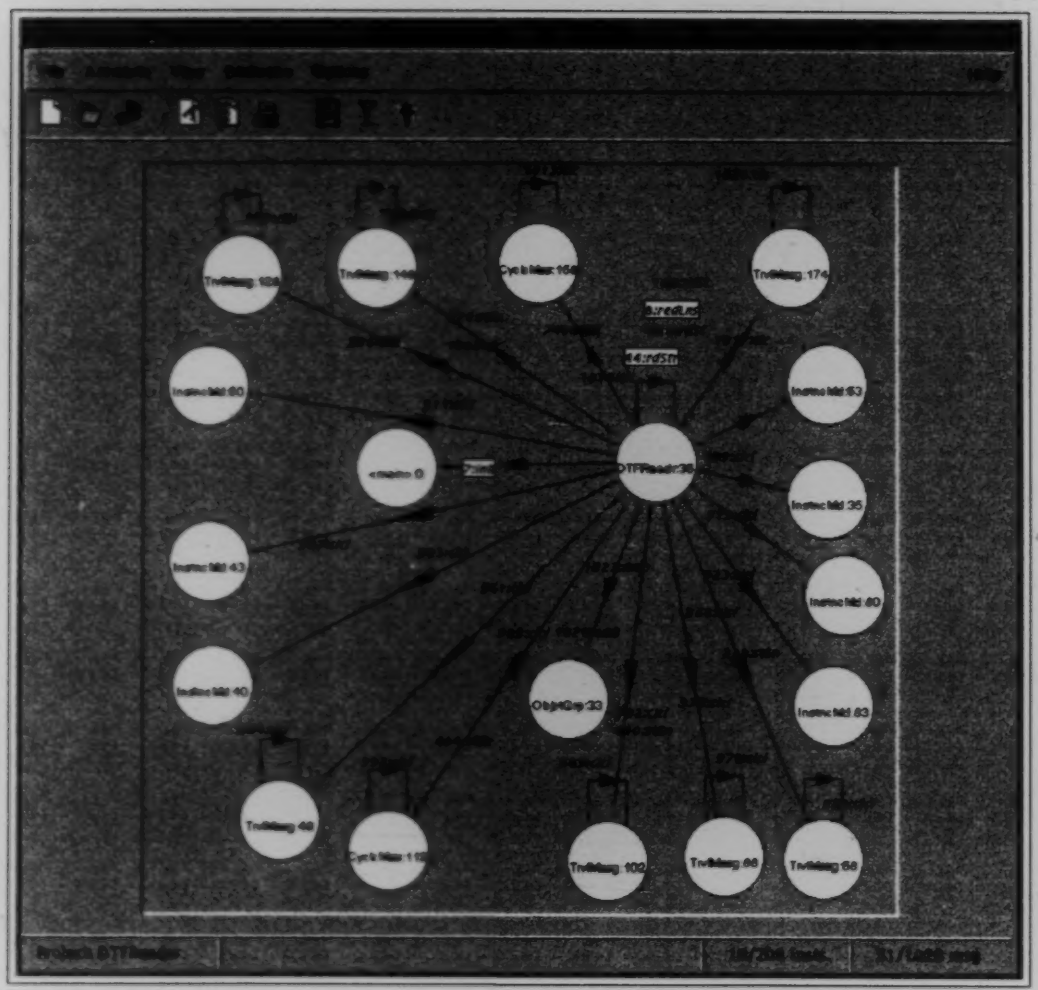


Figure 7.1: Object Graph of the DTFReader Application

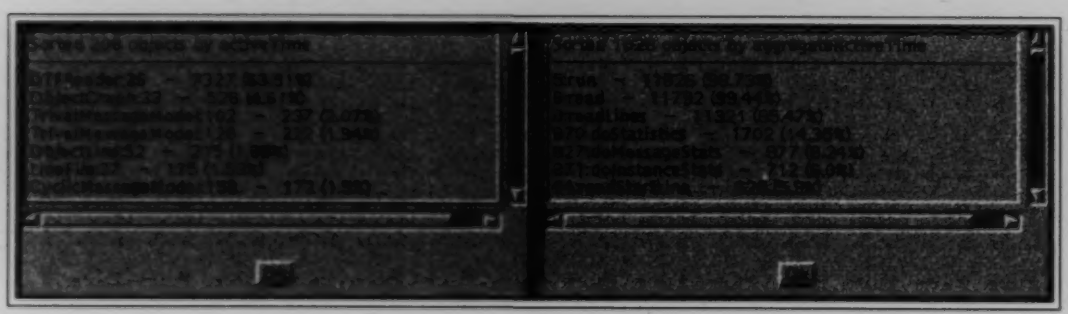


Figure 7.2: Runtime Statistics of the DTFReader Application

DTFReader												
	Sequential w/ JGT				Parallel w/ JGT				Parallel w/ JNT			
# of files	user time	sys. time	wall clk.	cpu usg.	user time	sys. time	wall clk.	cpu usg.	user time	sys. time	wall clk.	cpu usg.
1	9.0 s	0.0 s	0:10	91%	9.0 s	0.0 s	0:09	94%	9.0 s	0.0 s	0:10	93%
2	18.0 s	0.0 s	0:18	94%	18.0 s	0.0 s	0:18	97%	19.0 s	0.0 s	0:16	117%
4	36.0 s	0.0 s	0:36	97%	36.0 s	0.0 s	0:36	97%	45.0 s	3.0 s	0:31	154%

Table 7.1: Performance of the Parallel DTFReader

Table 7.1 presents the performance of a sequential DTFReader, a parallel version with JGT, and a parallel version with JNT. There are four observations that are worthy of discussion. First, the parallel version with JGT could not outperform the sequential version since only one processor could be used. Second, the performance is faster by about 11% for 2 files (18 seconds vs. 16 seconds) and 14% for 4 files (36 seconds vs. 31 seconds) in the parallel version with JNT. Third, the CPU utilization rates were over 100% in the 2- and 4-thread versions, which suggests that the threads were keeping the two CPUs busy at the same time. Fourth, in the case of four threads, the system time was greater than zero. This implies that the threads were competing for the two CPUs, and therefore time was spent on system calls, context switches, thread interrupts, and so on. For this example, Revy successfully helped us parallelize the application and obtain performance gains.

7.2.4 Probe Effects

The probe effects due to the annotation of the DTFReader were analyzed with different combinations of class and method instrumentation. Each of these combinations logged different numbers of instances and messages, which were plotted against the execution times (CPU time) of the program. For DTFReader, we had four different instrumentations: (1) Max - instrumented all classes and methods, (2) Half M – half of the methods in each class, (3) Half C – all the methods in half of the classes, and (4) Quarter – half of the methods in half of the classes. The performance of the better annotation was compared to that of the faster annotation (see Section 5.5.3). We present the timings of the DTFReader for a 26-kilobyte file in Table 7.2, and the data is plotted in Figure 7.3.

DTFReader					
	None	Quarter	Half C	Half M	Max
instances		628	628	2712	2713
messages		4941	11603	6592	15906
	1.55 s				
faster		15.0 s	31.0 s	21.0 s	45.0 s
better		15.0 s	32.0 s	22.0 s	45.0 s

Table 7.2: Timings of the Probe Effects of DTFReader

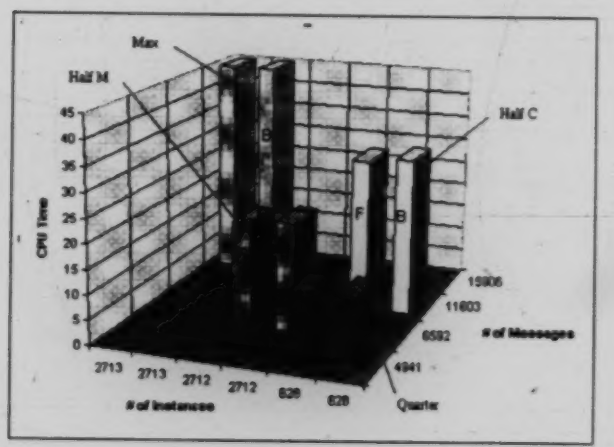


Figure 7.3: Plottings of the DTFReader Probe Effects

One can observe that the DTFReader application always generates many messages and instances, and instrumentation can slow down the program by a factor of 30. In general, this is too high. Hence, users should be more selective in the instrumentation of classes and methods in order to minimize the probe effects and obtain more accurate timings. Moreover, we observed that the better instrumentation did not slow down the application by much, as compared to the performance of the corresponding faster instrumentation. Therefore, users do not have to trade off accuracy with performance by using a faster instrumentation.

7.3 HTML File Parser

7.3.1 Program Description

The HTML file parser [KO96], code named HTMLParser, is an OO program based on a version written by one of the authors of the Java language. It takes multiple HTML files as input and checks whether or not the HTML syntax is correct. The application contains 16 classes and 141 public instance methods. We modified the source code slightly so that it initializes everything before looping to parse each input file. One can predict that there exists a high potential for the program to be parallelized. For instance, each call to the `parse()` method in the loop can be executed on a separate thread. Nevertheless, we employed Revy to carry out the analysis for us, to see if it could find this simple parallelization strategy.

7.3.2 Runtime Heuristics

We first applied full instrumentation to the HTMLParser program. Input was a 78-byte HTML file. However, the execution with these settings and input generated a huge runtime trace (3.5 megabyte in size, 777 instances and 23608 messages) which is too large to analyze using Revy. A brief investigation of the log files showed that the HTMLParser program created one object not only for each HTML tag, but also for each attribute list, each attribute value, and for the complete set of Document Type Definitions (DTDs). Hence, we instrumented only those classes and methods that are non-trivial and directly related to HTML parsing. The final instrumentation included 5 classes and 46 methods. Re-executing the code with a 4.3 KB HTML file generated a trace file of a reasonable size (234 KB). The trace file recorded 198 instances and 1443 messages. Figure 7.4 shows the call graph represented in the trace file. As before, we filtered out some objects, highlighted the most active ones and queried Revy about the statistics.

The statistics in Figure 7.5 show that the instance `Parser` and the messages `parse()` and `parseContent()` were the only good candidates for parallelization, as they were at the top of their lists and had active times over 1000 milliseconds.

However, by inspecting the message order numbers, we noticed that the message `4:parseContent()` followed the message `3:parse()` immediately, and in fact, the former was called within the latter. In addition, the time spent in `3:parse()` that was not spent in `4:parseContent()` was much smaller than 1000 milliseconds. Therefore, it is not efficient to execute the two methods on different threads. By spawning the message `parse()` on another thread, all the following method calls will be executed on the new threads.

The next step was to decide which one to make concurrent: should we parallelize the instance `Parser` or just its method `parse()`? The difference is, by creating a `Parser()` on a different thread, we would have all the called methods of this new object run on the new thread; whereas by spawning the method on a new thread only, all its following method calls would be executed on the new thread. By inspecting the un-filtered call graph of HTMLParser, we noticed that `Parser` was a relatively independent object, and most of its method calls were circular messages. Therefore, as suggested by the Revy system, we parallelized the application by creating each `Parser` object on a new thread. This observation fulfills our prediction stated before. The parallelization results are presented in the next section.

7.3.3 Parallelization Results

We parallelized the HTMLParser by making the class `Parser` a subclass of the `Thread` class in Java. Hence, each `Parser` instance created for each HTML file should be spawned on and mapped to a different processor, and thus it could parse the HTML file on its own. The performance was timed using 1, 2 and 4 HTML files of size 1 megabyte. Table 7.3 shows the performance results of the experiments. Once again, the performance of the parallel version using JGT is the same as that of the sequential with JGT. Unfortunately, no performance gain was achieved with either the two- or the four-thread version using JNT. There are three observations that are noteworthy. First, the one-thread version which used JNT had unexpectedly poor performance. It should be comparable to the performance of the sequential version using JGT. Second, the performance tends to decrease in an over-linear fashion as the number

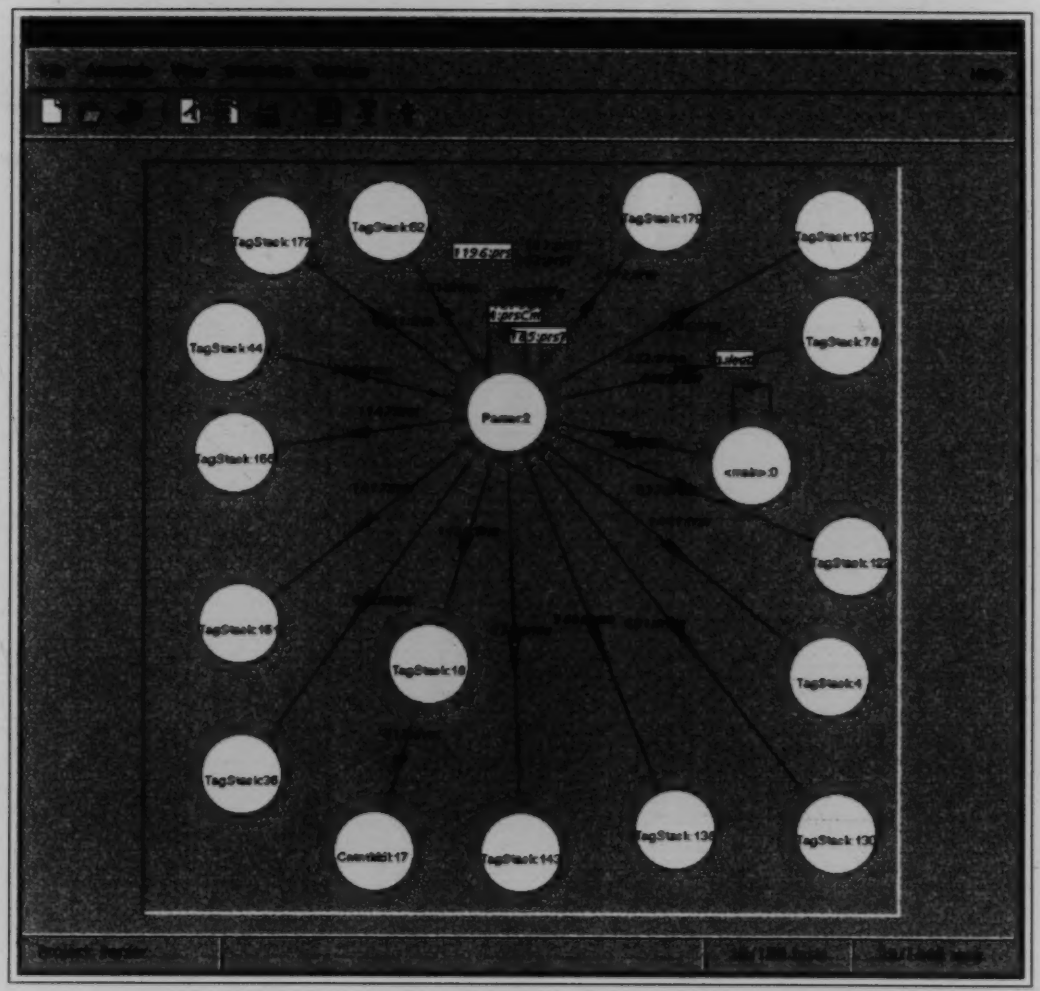


Figure 7.4: Object Graph of the HTMLParser Application

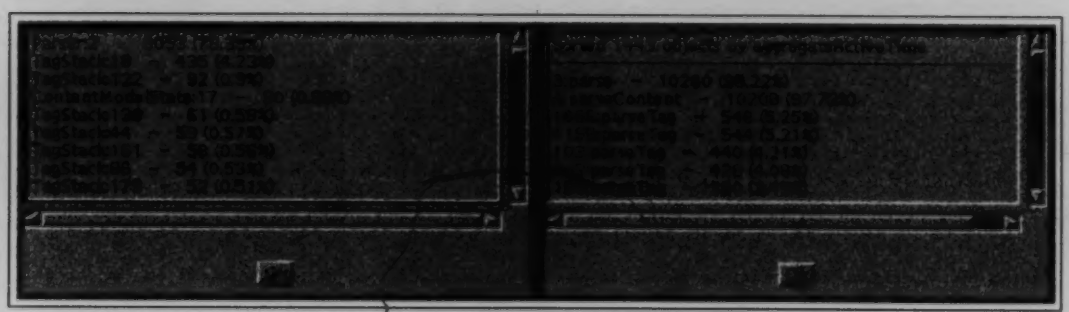


Figure 7.5: Runtime Statistics of the HTMLParser Application

HTMLParser												
	Sequential w/ JGT				Parallel w/ JGT				Parallel w/ JNT			
# of files	user time	sys. time	wall clk.	cpu usg.	user time	sys. time	wall clk.	cpu usg.	user time	sys. time	wall clk.	cpu usg.
1	15.0 s	0.0 s	0:16	90%	15.0 s	0.0 s	0:16	92%	29.0 s	0.0 s	0:29	97%
2	30.0 s	0.0 s	0:30	97%	30.0 s	0.0 s	0:31	96%	51.0 s	0.0 s	0:51	98%
4	60.0 s	0.0 s	1:01	97%	59.0 s	0.0 s	1:00	97%	150.0 s	59.0 s	2:13	156%

Table 7.3: Performance of the Parallel HTMLParser

of threads increased. Third, the timings of the four-thread version show that the parallel HTMLParser utilized both CPUs. One can see this from the fact that the CPU usage was over 100%, and the wall clock time was shorter than the total process time (the sum of the user time and system time).

The poor performance of the JNT version required some investigations. Using the `mpstat`[3] command showed that the application spent a large amount of time on thread interrupts, context switches and system calls. Hence, the nature of this application induced a great number of hidden system calls in the Java runtime library. Moreover, we noticed that the HTMLParser created a large number of instances even with a small HTML file. For example, the application created 777 instances with a 78-byte HTML file. Therefore, we argue that the poor performance was due to two reasons: (1) hidden system calls in the Java library, and (2) excessive numbers of created instances. These hypotheses were tested, and the results are presented in Section 7.5.

7.3.4 Probe Effects

The HTMLParser's probe effects were measured with four different combinations of class and method instrumentation, including: (1) Max - only non-trivial and HTML-parsing related classes and methods (i.e. the 5 classes and 46 methods mentioned in Section 7.3.2), (2) Half M - half of the methods in each class in Max, (3) Half C - all

[3]The `mpstat` command is a Solaris utility program which reports per-processor statistics, such as number of interrupts, context switches, spins on mutexes, system calls, and I/O wait, user time and processor idle percentages.

HTMLParser					
	None	Quarter	Half C	Half M	Max
instances		334	334	1887	1887
messages		3677	13029	5970	14888
	0.90 s				
faster		10.0	32.0 s	18.0 s	39.0 s
better		10.0	33.0 s	18.0 s	40.0 s

Table 7.4: Timings of the Probe Effects of HTMLParser

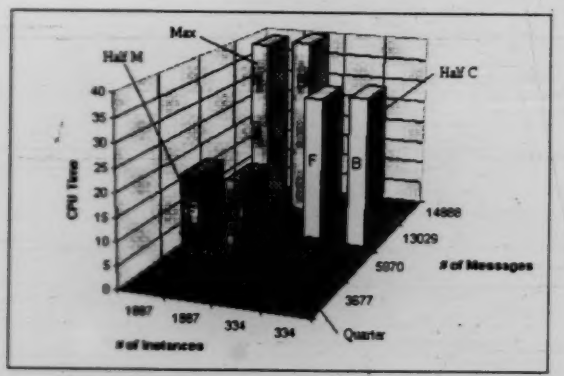


Figure 7.6: Plottings of the HTMLParser Probe Effects

the methods in half of the classes in Max, and (4) Quarter – half of the methods in half of the classes in Max. We applied the instrumented application to a 11-kilobyte HTML file. The timing results are presented in Table 7.4, and the performance data is plotted in Figure 7.6. Similar to DTFReader, an HTMLParser instance always generates many messages and instances, and instrumentation can slow down the application by a factor of 45. Users therefore have to carefully select the classes and methods they want to instrument in order to obtain more accurate timings. Moreover, we observed that the better instrumentation was almost as efficient as the faster one. Therefore, users do not have to trade off accuracy with performance by using a faster instrumentation.

7.4 A Zip Application

7.4.1 Program Description

The third application tested was a zip program, called JHLZip [LE96]. The program creates a `.zip` file without any compression. Hence it is similar to the UNIX `tar` utility program. The only difference is that it conforms to the zip standard proposed by PKWARE Incorporated. The standard requires any zip program to implement a checksum algorithm which guarantees that the zip file is intact. The JHLZip program is composed of 7 classes and 27 public instance methods. The application takes a text file as input which names the files to be zipped, and produces a `.zip` file. Since it is a zip program, one can expect that it is computationally intensive as compared to the previous two applications.

7.4.2 Runtime Heuristics

We instrumented all the classes and methods of the JHLZip package, and compiled and executed it with four 1-megabyte text files. The execution produced a small trace file of 8 kilobytes. This is because this program is less "object-oriented" than the previous two, so it creates fewer instances and invokes less messages. The object call graph represented by the trace file is shown in Figure 7.7. It has 16 instances and 43 messages recorded during runtime. The graph was re-arranged, and the four most active instances and messages were highlighted. The runtime statistics of the objects and messages are shown in Figure 7.8.

The statistics show that the most active instances are `ZipFile` and the four `CyclicRedundancyCheck` instances, and `write()` and `update32()` are the most active messages. We noticed from the graph that the `update32()` messages (message IDs: 29, 32, 35 and 38) always follow `write()` messages (message IDs: 28, 31, 34 and 37). However, the inspector windows on the messages revealed that the two methods belong to two different classes `ZipFileHeader` and `CyclicRedundancyCheck`, respectively. Therefore in this case, spawning a `ZipFileHeader` instance remotely while keeping the `CyclicRedundancyCheck` instance local will lead to two remote mes-

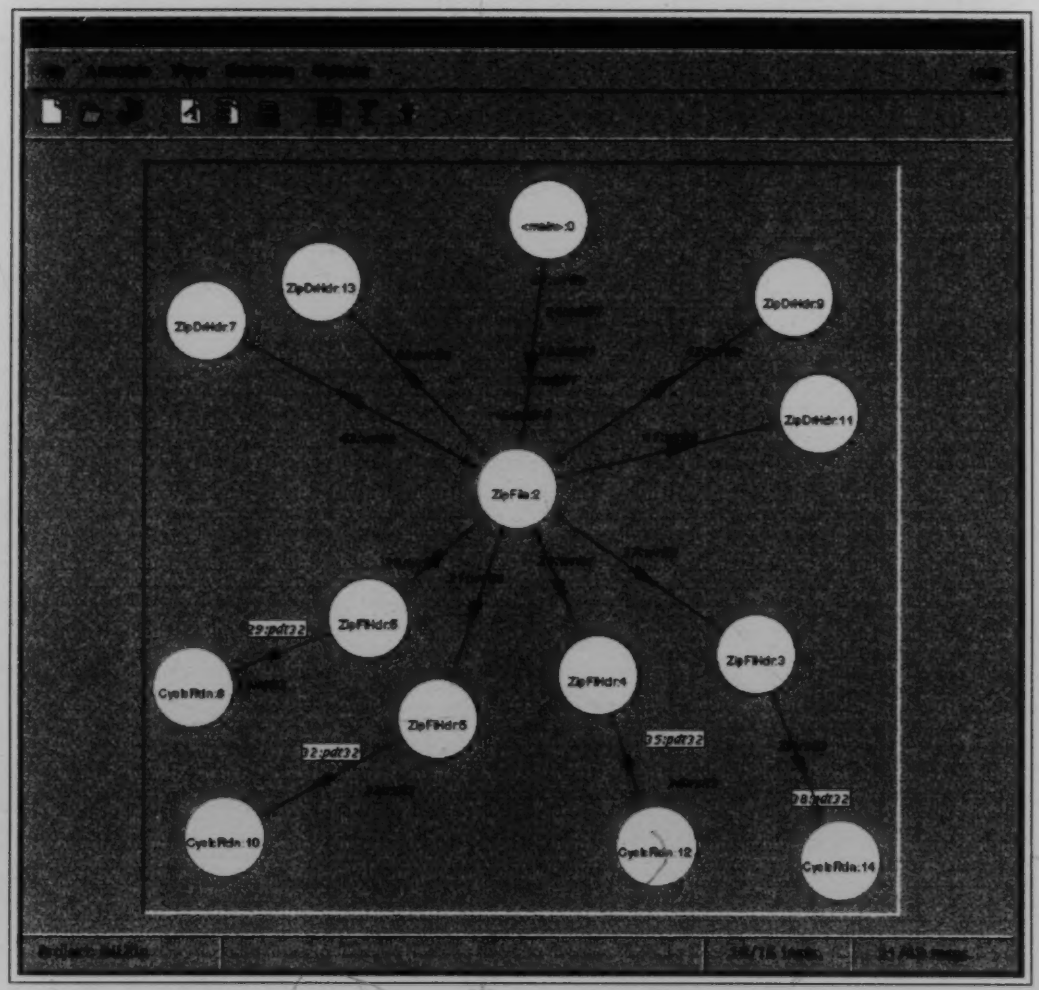


Figure 7.7: Object Graph of the JHLZip Application

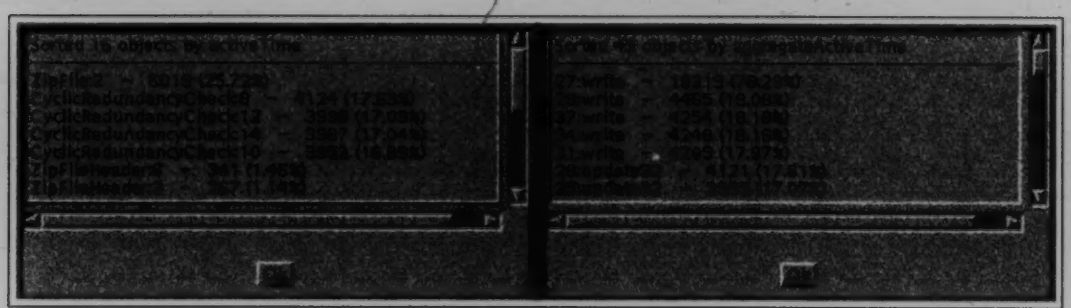


Figure 7.8: Runtime Statistics of the JHLZip Application

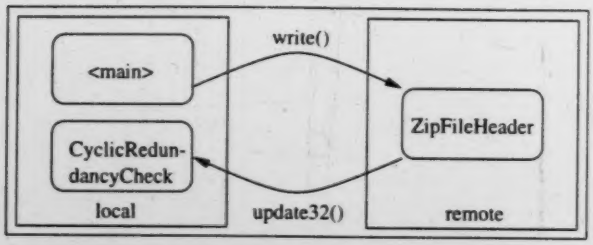


Figure 7.9: Parallelizing `ZipFileHeader`

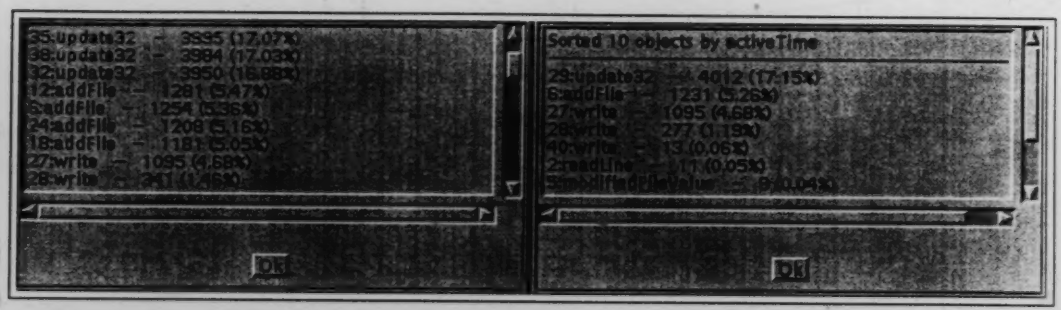


Figure 7.10: Net and Average Active Times of the Messages of JHLZip

sages: one `write()` and one `update32()` (see Figure 7.9). In addition, we observed that the `ZipFileHeader` objects were not very active (less than 400 milliseconds) and thus it is unnecessary to create it remotely. Moreover, the net and the average active times of the `write()` messages were short as compared to those of the `update32()` messages (Figure 7.10). As a result, we can deduce that, out of the two methods `write()` and `update32()`, only the latter one needs to be executed remotely.

For the most active instances, `ZipFile` and `CyclicRedundancyCheck`, we decided not to spawn the `ZipFile` instance remotely. This is because the object graph shows that the `ZipFile` instance was the central object and thus we would have introduced too many remote calls if the instance was created remotely while the rest of the instances remained local. As a result, we decided to put `CyclicRedundancyCheck` instances on new threads as each of them had only one incoming message, `update32()`, which was expensive enough to be parallelized.

JHLZip												
	Sequential w/ JGT				Parallel w/ JGT				Parallel w/ JNT			
# of files	user time	sys. time	wall clk.	cpu usg.	user time	sys. time	wall clk.	cpu usg.	user time	sys. time	wall clk.	cpu usg.
1	14.0 s	1.0 s	0:16	88%	14.0 s	1.0 s	0:16	88%	14.0 s	1.0 s	0:15	94%
2	27.0 s	2.0 s	0:31	91%	27.0 s	2.0 s	0:31	91%	27.0 s	2.0 s	0:18	154%
4	54.0 s	3.0 s	1:01	91%	54.0 s	3.0 s	1:01	93%	54.0 s	3.0 s	0:35	158%

Table 7.5: Performance of the Parallel JHLZip

7.4.3 Parallelization Results

The `CyclicRedundancyCheck` instances were parallelized by subclassing them under the `Thread` class in Java. We tested the application with one, two and four 7.5 MB input files and compared the results with the same settings running sequentially. The experimental results are presented in Table 7.5. As before, the performance of the sequential and parallel versions with JGT are the same. However, there is an obvious performance gain when two or more files were zipped concurrently using JNT. For instance, in the case of four files, the application obtained a reduction in execution time of 26 seconds on the wall clock measurements (from 61 seconds with JGT to 35 seconds with JNT). Moreover, we noticed that the utilization rate of the CPUs was over 100%. This implies that all the virtual threads were mapped to both CPUs, which were being kept busy during the course of execution. The performance gains are 41.9% and 42.6% in the 2-thread and 4-thread versions, respectively. The gains are less than 50% because there were overheads spent on the Java Virtual Machine with JNT. The performance gain is larger than that of the parallel DTFReader with JNT. The reason is that DTFReader contains more object manipulation method calls, while JHLZip is a computationally intensive application. As a result a thread can run more efficiently in the latter case. In fact, these are exactly the type of applications which we want to parallelize.

In addition, we implemented another parallel version which executed the method `update32()` remotely. We used the `MethodThread` package [MA97] from Steve MacDonald. The `MethodThread` package uses the Java reflection capabilities to create threads that execute an arbitrary method on an object. By providing method

JHLZip w/ MethodThread								
	Parallel w/ JGT				Parallel w/ JNT			
# of files	user time	sys. time	wall clk.	cpu usg.	user time	sys. time	wall clk.	cpu usg.
1	14.0 s	1.0 s	0:16	89%	14.0 s	1.0 s	0:15	68%
2	27.0 s	2.0 s	0:31	88%	27.0 s	2.0 s	0:18	150%
4	54.0 s	3.0 s	1:01	92%	54.0 s	3.0 s	0:35	158%

Table 7.6: Performance of the JHLZip Parallelized with MethodThread

spawning[4], it enriches Java's multithreading protocol which only allows objects to be threaded. However, despite the fact that the package provides us with many protocols (constructors) to create a method thread, the programming in our case was not as easy as subclassing the `CyclicRedundancyCheck` class. For instance, we had to introduce a new class and write four lines of code with `try-catch` statement (versus two lines in the previous case) in order to spawn one single method. Nevertheless, we were able to obtain performance gains. The results are presented in Table 7.6. One can see that this version has roughly the same performance as the previous parallel version.

7.4.4 Probe Effects

The probe effects of JHLZip were similarly measured with four combinations of class and method instrumentation. They are: (1) Max – all classes and methods, (2) Half M – half of the methods in each class, (3) Half C – all the methods in half of the classes, and (4) Quarter – half of the methods in half of the classes. Since the application runs very fast with small files, we used a much larger data input. We applied the instrumented application with 32 one-megabyte text files. The timings are presented in Table 7.7, and the data is plotted in Figure 7.11. The degradation in performance is negligible compared to the original code. This clearly shows that: (1) this program is computationally intensive, and (2) very few instances were created and few messages were invoked. Therefore, JHLZip is not a "true" OO application, so users can always apply full or better instrumentation without consequences.

[4]As a matter of fact, method spawning is not new in OO languages. Smalltalk provides a user-friendly syntax to "fork" off as many blocks of code as the user wants.

JHLZip					
	None	Quarter	Half C	Half M	Max
instances		36	36	101	101
messages		130	227	162	324
	69.0 s				
faster		69.0	70.0 s	70.0 s	71.0 s
better		69.0	70.0 s	70.0 s	71.0 s

Table 7.7: Timings of the Probe Effects of JHLZip

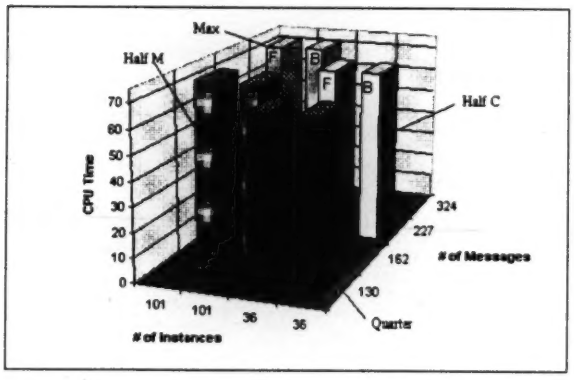


Figure 7.11: Plottings of the JHLZip Probe Effects

7.5 An Object Adder

The poor performance observed for the parallel HTMLParser with JNT requires a more detailed explanation. The reasons suggested in Section 7.3.3 are: (1) hidden system calls in the Java library, and (2) excessive number of instances created. We developed a test program, ObjectAdder, to measure the effects of method synchronization and instance creation in Java.

7.5.1 Program Description

ObjectAdder is a simple adding program, which adds the numbers from 1 to 3,000,000. Figure 7.12 shows a partial listing of the code. This program does not simply loop and add, but it calls one of four adding methods in each iteration of the loop. The

```
switch(whatToRun)
{
  case NOSYNC_INT:
    for(i=0 ; i<MAX ; i++) this.syncAddInt(i) ;
    break ;
  case SYNC_INT:
    for(i=0 ; i<MAX ; i++) this.nosyncAddInt(i) ;
    break ;
  case NOSYNC_INTEGER:
    for(i=0 ; i<MAX ; i++) this.syncAddInteger(new Integer(i)) ;
    break ;
  case SYNC_INTEGER:
    for(i=0 ; i<MAX ; i++) this.nosyncAddInteger(new Integer(i)) ;
    break ;
}

public void nosyncAddInt(int i)
  { this.sum += i ; }
public synchronized void syncAddInt(int i)
  { this.sum += i ; }
public void nosyncAddInteger(Integer i)
  { this.sum += i.intValue() ; }
public synchronized void syncAddInteger(Integer i)
  { this.sum += i.intValue() ; }
```

Figure 7.12: The ObjectAdder Program

four adding methods are critical to the outcomes. They are:

- regular, non-synchronized, `int` adder,
- synchronized `int` adder,
- regular, non-synchronized, `Integer` object adder, and
- synchronized `Integer` object adder.

We believe that having a large number of instances has a high impact on the performance of an OO program. In addition, the use of synchronization also slows down a program because further analysis of the HTMLParser application showed that the application had many system calls and thread interrupts. As a matter of fact,

it uses many `Vector` and `Hashtable` objects whose member functions are mostly synchronized. There are 36 test cases – four methods versus three different numbers of threads versus three versions of code. The experimental results and analysis are presented in the following section.

7.5.2 Performance Analysis

The 36 test results are organized into 3 tables, according to the number of threads spawned. They are presented in Table 7.8, Table 7.9 and Table 7.10. Note that all the tests had the memory effects eliminated since we pre-allocated a memory pool of 32 megabytes to the heap and disabled the asynchronous garbage collector. As before, the parallel performance using JGT is roughly the same as the sequential performance using JNT. In addition, the 36 tests led to 10 observations:

1. JNT can outperform JGT on the synchronized `int` adders only.
2. For each version of the number of threads, the synchronized `int` adders with JNT always have the highest CPU utilization rate and the best performance.
3. Increasing the number of threads degrades the JNT performance.
4. With JNT, more threads lead to longer system time, which includes the I/O, system wait, system call, thread interrupt times, and so on.
5. More threads lead to a higher system-user time ratio on JNT.
6. Increasing the number of threads increases the CPU utilization rate on both JNT and JGT.
7. The synchronized adders always have a slower performance than non-synchronized adders.
8. The `Integer` adders always run longer.
9. The `Integer` adders always have a higher impact on the performance than the synchronized adders do.

10. The synchronized adders always have a higher impact on the performance with JNT than JGT.

All the above observations can be traced to one root cause: synchronization with the internal object table in a Java Virtual Machine (VM). The invocation of a synchronized method will put a lock on the receiver object. In other words, the Java Virtual Machine will look up the entry corresponding to the receiver object from the object table and lock it. Moreover, an instance creation may slow down the system by allocating memory and performing garbage collection. However, we avoided these effects in the settings of our experiments. Thus, the problem occurring behind the scene of an instance creation can again be explained by the synchronization with the internal object table. When an instance is being created, the Java VM has to lock the whole object table before inserting an entry of the new object in the table. Hence, the object table is synchronized as well.

We also observed that the `Integer` adders always have a higher impact on the performance than the synchronized adders do. The performance table in [HA97] also confirms this observation. This can be explained by the fact that the process invoked by a synchronized message obtains a read lock only on the object table, while an instance creation will lead to an exclusive write lock on the object table. As a result, an application will perform slower in the case of instance creation in general. In addition, the performance is further degraded with the use of JNT. This is due to the fact that the kernel threads, which reside on different processors, share one global object table. Therefore, it involves a more complex system-level synchronization mechanism to lock or unlock the object table.

Several articles [LA96, YO96, BE97, HA97, SS97] proposed various ways to optimize a Java application. Some of the optimization techniques for Java programming are summarized below. First, `Vector` and `Hashtable` objects are inefficient. Arrays should be used instead if the structure size is known beforehand. Second, unnecessary instance creation should be avoided. For example, primitive numeric data should be used instead of `Number` objects if possible. In addition, as mentioned in [SS97], the

One Thread												
	Sequential w/ JGT				Parallel w/ JGT				Parallel w/ JNT			
	user time	sys. time	wall clk.	cpu usg.	user time	sys. time	wall clk.	cpu usg.	user time	sys. time	wall clk.	cpu usg.
n, i	4.0 s	0.0 s	0:05	71%	4.0 s	0.0 s	0:05	74%	4.0 s	0.0 s	0:05	73%
s, i	8.0 s	0.0 s	0:09	87%	8.0 s	0.0 s	0:09	88%	30.0 s	0.0 s	0:31	95%
n, I	14.0 s	0.0 s	0:14	93%	13.0 s	0.0 s	0:14	93%	27.0 s	0.0 s	0:28	96%
s, I	16.0 s	0.0 s	0:17	92%	17.0 s	0.0 s	0:18	90%	58.0 s	0.0 s	0:59	98%
n – no sync., s – sync., i – `int`, I – `Integer`												

Table 7.8: Performance of the ObjectAdder with One Thread

Two Threads												
	Sequential w/ JGT				Parallel w/ JGT				Parallel w/ JNT			
	user time	sys. time	wall clk.	cpu usg.	user time	sys. time	wall clk.	cpu usg.	user time	sys. time	wall clk.	cpu usg.
n, i	9.0 s	0.0 s	0:09	93%	9.0 s	0.0 s	0:09	90%	9.0 s	0.0 s	0:05	171%
s, i	16.0 s	0.0 s	0:16	95%	16.0 s	0.0 s	0:16	93%	57.0 s	1.0 s	0:57	100%
n, I	28.0 s	0.0 s	0:28	97%	27.0 s	0.0 s	0:27	97%	96.0 s	71.0 s	2:09	128%
s, I	32.0 s	0.0 s	0:33	95%	34.0 s	0.0 s	0:35	95%	197.0 s	136.0 s	4:26	124%
n – no sync., s – sync., i – `int`, I – `Integer`												

Table 7.9: Performance of the ObjectAdder with Two Threads

statement `s = new String(''Hello'')` is redundant, as it creates two `String` instances. It should simply be written as `s = ''Hello''`[5].

In summary, we experimented and studied the performance hits in Java due to synchronized messages and instance creation. In addition, the Java optimization techniques are presented. The above problems in Java also exist in `DTFReader`. As a result, we suggest a re-design and re-implementation of the module with the above guidelines in mind.

7.6 Guidelines for Parallelization

In this section, we describe some guidelines for OO program parallelization, based on the experience of using Revy. Before applying any parallelization, programmers should understand the architecture and the cost of communication. Since shared

[5]This redundancy is suggested in the white paper from Sun Microsystems. It makes no sense that they do not remove it from their Java programming interface.

Four Threads												
	Sequential w/ JGT				Parallel w/ JGT				Parallel w/ JNT			
	user time	sys. time	wall clk.	cpu usg.	user time	sys. time	wall clk.	cpu usg.	user time	sys. time	wall clk.	cpu usg.
n, i	17.0 s	0.0 s	0:18	92%	17.0 s	0.0 s	0:18	92%	17.0 s	0.0 s	0:09	174%
s, i	32.0 s	0.0 s	0:32	97%	32.0 s	0.0 s	0:32	95%	158.0 s	51.0 s	2:38	131%
n, I	55.0 s	0.0 s	0:56	97%	55.0 s	0.0 s	0:55	98%	237.0 s	223.0 s	5:09	148%
s, I	64.0 s	0.0 s	1:05	98%	66.0 s	0.0 s	1:07	98%	570.0 s	567.0 s	12:33	150%
n – no sync., s – sync., i – `int`, I – `Integer`												

Table 7.10: Performance of the ObjectAdder with Four Threads

memory machines incur no (or minimal) communication cost, multiprocessor machines require different parallelization analysis than networks of workstations where the cost of communication can be very high. In both cases, parallel programmers should at least know the number of CPUs their applications will use.

For non-shared memory machines, there are two guidelines that do not depend on the Revy statistics. First, the user should identify the environment-dependent objects (for example, I/O objects). These objects are hardware dependent and may have to reside on specific workstations. Second, any object that has only one neighbour can be assigned to the same workstation to avoid unnecessary communication costs.

The first general guideline is that only those instances/messages that have an active time over a specific threshold should be considered for parallelization. For shared memory machines, there is virtually no communication cost, and a programmer can spawn as many threads as the system supports. However, threads still have a creation cost, and incur system costs such as context switches and thread interrupts. Based on the applications studied in this thesis, the threshold for the shared memory machine used is about 1000 milliseconds. This threshold may be different on other hardware or when using a different Java interpreter or other language. The threshold for networks of workstations depends on the communication costs of the network.

The second general guideline is to start with the top instances/messages and work down until the minimum granularity is reached or until there is enough parallelism in

the application. Clearly, if there are N processors, we want at least N tasks running in parallel (and usually, there will be additional performance gains by having multiple threads/processes per processor).

The third general guideline is based on the difference between average and aggregate activity. Aggregate activity describes the performance of an individual instance/message. Average activity represents the overall performance. By examining aggregate activity a programmer can tell whether to parallelize all the instances of that class/method, or just an individual instance/message.

The fourth general guideline is based on successive message sends. In the case of method parallelization, a programmer should inspect the net activity and the message order, and study the call sequence by means of object call graph animation to decide where to split a message chain. For example, assume the two most active messages listed in the statistics involve one message as an internal message sent by the other. In this case, a programmer must study the net message times to decide whether it is worthwhile to spawn each message on a new separate thread, or to only spawn one additional new thread that executes both messages.

The fifth and final general guideline is that objects with many cyclic messages are good candidates for parallelization. Such objects are likely to have high activity level. They are also less dependent objects that are loosely connected to the rest of the system. By spawning them on new threads, programmers will obtain higher utilization rates and less communication costs.

In conclusion, finding a good parallel decomposition of a program is often a matter of experience. Although guidelines are useful as a starting point, the variety of applications gives rise to numerous exceptions to any program parallelization methodology. Unfortunately, the state of the art of automating this aspect is still a long way off into the future.

7.7 Summary

This chapter shows how Revy has been applied to different applications, discusses the suggestions given in the runtime heuristics, and analyzes the performance gain (or loss) for each application after parallelization. Moreover, the probe effects due to the extra code added by instrumentation and the performance on synchronized methods and instance creation in Java are studied. In summary, this chapter shows that Revy is useful in identifying the parallelization bottleneck of Object-Oriented programs.

Chapter 8

Conclusion

This chapter concludes the thesis with a review of some improvements and future research identified for the three subsystems of Revy. The current implementation should provide a framework for this work.

8.1 Future Work

8.1.1 Visualization and Interaction Subsystem

The Visualization and Interaction Subsystem needs a few research improvements. Future work should focus on designing and implementing techniques to reduce the cluttering problems of object call graphs. In Chapter 4, we suggested a few approaches such as grouping instances and messages, zooming in/out, auto-filtering and source code animation. Moreover, we should also consider implementing the graph visualization technique, Hyperbolic Display, which smoothes the blending between the focus and the context of a complex call graph.

8.1.2 Parsing and Annotation Subsystem

The Parsing and Annotation Subsystem is the most mature component of Revy. There are not many areas that need to be improved. We could store complete information about the user and system classes during the parsing phase. We could also add instrumentation code for handling exceptions. As well, the `C++Sourcer`, which parses and annotates C++ programs, could be modified to support some requirements that

were added after the first implementation. Such modifications would provide more evidence of the language neutrality of Revy. Revy would also benefit from replacing the source-to-source translator Sage++ by a more mature and robust tool, such as SUIF [SU97].

8.1.3 Runtime Modeling Subsystem

In RMS, there are two major areas that future research could focus on. First, in order to fulfill the goal of being a parallelization advisor, Revy could implement the parallelization algorithms suggested in Section 6.5.3. Since there are conceptual differences between parallelizing object-oriented systems and parallelizing procedural ones, we should research on heuristics which optimally map objects and methods to processors. Second, Revy could include an auto parallelization feature. After PAS has identified the high-granularity objects/methods, it could automatically insert threads into the user's code by using either regular `Thread` class for high granularity objects or `MethodThread` class for high granularity methods.

A few optimizations could be applied to the current Runtime Modeling Subsystem. `DTFReader` is the bottleneck of the system, and it should be re-designed and re-implemented to read a DTF file asynchronously. In addition, more arrays and primitive data types should be used instead of inefficient objects such as `Vector` and `Integer`. Moreover, we could enhance the runtime statistics, such as communication and processing costs. We could also enrich the cost model by considering the number of machines, communication cost per byte between two machines, relative processing costs per unit of time, and so on.

8.2 Summary

This thesis showed how visualization and granularity can be combined together to help program parallelization. It argued that, in addition to function-based profiling data, instance-based statistics and runtime interaction diagrams are essential for

programmers to transform their sequential object-oriented programs to parallel ones. The result of the thesis is a platform-neutral, easy-to-use, object/method granularity visualization system called Revy. There are three subsystems in Revy: Visualization and Interaction, Parsing and Annotation, and Runtime Modeling. The thesis defined the requirements of Revy as well as its subsystems. Moreover, it presented Revy's modular architecture, OO design and implementation, and discussed the outstanding issues of the subsystems. The usefulness of the tool was justified by performing a few experiments to aid program parallelization. Finally, we identified some future research areas and improvements for enhancing Revy.

APPENDIX A: Original User's Code

```
/* ********************** Main.java ********************** */

public class Main
{
   public static void main(String argv[])
   {
     Person will, duane, jonathan ;
     Account willAcct, duaneAcct, jonathanAcct ;


     will = new Person("Will") ;
     willAcct = new Account(1000.0) ;
     will.setAccount(willAcct) ;

     duane = new Person("Duane") ;
     duaneAcct = new Account(1000.0) ;
     duane.setAccount(duaneAcct) ;

     jonathan = new Person("Jonathan") ;
     jonathanAcct = new Account(1000.0) ;
     jonathan.setAccount(jonathanAcct) ;

     duaneAcct.transfer(100.0, willAcct) ;
     jonathanAcct.transfer(100.0, willAcct) ;

     System.out.println(will) ;
     System.out.println(duane) ;
     System.out.println(jonathan) ;
   }
}


/* ********************** Person.java ********************** */

public class Person extends Object
{
   public static int SIN = 123456789 ;
   public int sin ;
   public String name ;
   public Account account ;


   public Person(String n)
   {
     this.sin = SIN++ ;
     this.name = n ;
   }

   public String toString()
   {
     return("<Person sin=" + this.sin + " name=" + this.name + "> /n"
         + account.toString()) ;
   }

   public int getSIN() { return(this.sin) ; }
   public String getName() { return(this.name) ; }
   public Account getAccount() { return(this.account) ; }
   public void setAccount(Account acct) { this.account = acct ; } }

/* ********************** Account.java ********************** */

public class Account extends Object
{
```

```
    public static int ACCOUNTNUMBER = 654321 ;
    public int accountNumber ;
    public double balance ;


    public Account(double amt)
    {
      this.accountNumber = ACCOUNTNUMBER++ ;
      this.balance = amt ;
    }

    public String toString()
    {
      return("<Account AccoutNumber=" + this.accountNumber +
          " balance=" + this.balance + ">") ;
    }

    public int getAccountNumber() { return(this.accountNumber) ; }

    public double getBalance() { return(this.balance) ;

    public void deposit(double amt)
    {
      this.balance += amt ;
    }

    public boolean withdraw(double amt)
    {
      if(this.balance < amt)
      {
        System.err.println("Sorry, not enough money for withdrawal") ;
        return(false) ;
      }
      this.balance -= amt ;
      return(true) ;
    }

    public boolean transfer(double amt, Account acct)
    {
      if(this.withdraw(amt))
      {
        acct.deposit(amt) ;
        return(true) ;
      }
      return(false) ;
    }
}
```

APPENDIX B: User's Code with Exact Instrumentation

```
/* ********************** Main.java ********************** */

// REVY imported classes
import java.util.Hashtable ;
import whui.revy.util._RvObject ;
import whui.revy.util._RvLogger ;


public class Main implements _RvObject
{
   public static void main ( String argv [ ] )
   {
     _RvLogger._rvLogInit(".", "Main") ;
     {
       Person will , duane , jonathan ;
       Account willAcct , duaneAcct , jonathanAcct ;
       will = new Person ( "Will" ) ;
       willAcct = new Account ( 1000.0 ) ;
       will.setAccount ( willAcct ) ;
       duane = new Person ( "Duane" ) ;
       duaneAcct = new Account ( 1000.0 ) ;
       duane.setAccount ( duaneAcct ) ;
       jonathan = new Person ( "Jonathan" ) ;
       jonathanAcct = new Account ( 1000.0 ) ;
       jonathan.setAccount ( jonathanAcct ) ;
       duaneAcct.transfer ( 100.0 , willAcct ) ;
       jonathanAcct.transfer ( 100.0 , willAcct ) ;
       System.out.println ( will ) ;
       System.out.println ( duane ) ;
       System.out.println ( jonathan ) ;
     }
     _RvLogger._rvLogDone("Main") ;
   }

   // REVY added 1 field and 5 methods
   private int _rvOID = 0 ;
   public String _rvClassName() { return("Main") ; }
   public int _rvGetOID() { return(_rvOID) ; }
   public void _rvSetOID(int _i) { _rvOID = _i ; }
   public int _rvStaticSize() { return(0) ; }
   public int _rvRuntimeSize(Hashtable _vset)
   {
     int _acc = 0 ;
     if(_vset.contains(this)) { return(4) ; }
     _vset.put(this, this) ;
     return(_acc) ;
   }
}


/* ********************** Person.java ********************** */

// REVY imported classes
import java.util.Hashtable ;
import whui.revy.util._RvObject ;
import whui.revy.util._RvLogger ;


public class Person extends Object implements _RvObject
{
   public static int SIN = 123456789 ;
   public int sin ;
   public String name ;
```

```
    public Account account ;
    public Person ( String n )
    {
      this.sin = SIN ++ ;
      this.name = n ;
      _RvLogger._rvLogCreate(this) ;
    }
    public String toString ( )
    {
      return ( "<Person sin=" + this.sin + " name=" + this.name + ">/n" + account.toString ( ) ) ;
    }
    public int getSIN ( ) { return ( this.sin ) ; }

    public String getName ( ) { return ( this.name ) ; }

    public Account getAccount ( ) { return ( this.account ) ; }

    public void setAccount ( Account acct ) { this.account = acct ; }

    // REVY added 1 field and 5 methods
    private int _rvOID = 0 ;
    public String _rvClassName() { return("Person") ; }
    public int _rvGetOID() { return(_rvOID) ; }
    public void _rvSetOID(int _i) { _rvOID = _i ; }
    public int _rvStaticSize() { return(56) ; }
    public int _rvRuntimeSize(Hashtable _vset)
    {
      int _acc = 0 ;
      if(_vset.contains(this)) { return(4) ; }
      _vset.put(this, this) ;
      _acc += 4 ; // SIN
      _acc += 4 ; // sin
      _acc += 32 ; // name
      _acc += ((account == null)?  _NULL_SZE:account._rvRuntimeSize(_vset)) ; // account
      return(_acc) ;
    }
}


/* ********************* Account.java ********************* */

// REVY imported classes
import java.util.Hashtable ;
import whui.revy.util._RvObject ;
import whui.revy.util._RvLogger ;


public class Account extends Object implements _RvObject
{
   public static int ACCOUNTNUMBER = 654321 ;
   public int accountNumber ;
   public double balance ;
   public Account ( double amt )
   {
     this.accountNumber = ACCOUNTNUMBER ++ ;
     this.balance = amt ;
     _RvLogger._rvLogCreate(this) ;
   }
   public String toString ( )
   {
     return ( "<Account AccoutNumber=" + this.accountNumber + " balance=" + this.balance + ">" ) ;
   }
   public int getAccountNumber ( ) { return ( this.accountNumber ) ; }

   public double getBalance ( ) { return ( this.balance ) ; }

   public void deposit ( double amt )
   {
```

```
    int _sze = 0 ;
    Hashtable _vset = new Hashtable(32) ;
    _sze += 8 ; // amt
    _RvLogger._rvLogStart(this, "deposit", "!double^0!", _sze) ;
    {
      this.balance += amt ;
    }
    _RvLogger._rvLogEnd(this, "deposit", "!double^0!") ;
  }
  public boolean withdraw ( double amt )
  {
    int _sze = 0 ;
    Hashtable _vset = new Hashtable(32) ;
    _sze += 8 ; // amt
    _RvLogger._rvLogStart(this, "withdraw", "!double^0!", _sze) ;
    {
      if ( this.balance < amt )
      {
        System.err.println ( "Sorry, not enough money for withdrawal" ) ;
        _RvLogger._rvLogEnd(this, "withdraw", "!double^0!") ;
        return ( false ) ;
      }
      this.balance -= amt ;
      _RvLogger._rvLogEnd(this, "withdraw", "!double^0!") ;
      return ( true ) ;
    }
  }
  public boolean transfer ( double amt , Account acct )
  {
    int _sze = 0 ;
    Hashtable _vset = new Hashtable(32) ;
    _sze += 8 ; // amt
    _sze += ((acct == null)?  _NULL_SZE:acct._rvRuntimeSize(_vset)) ; // acct
    _RvLogger._rvLogStart(this, "transfer", "!double^0!Account^0!", _sze) ;
    {
      if ( this.withdraw ( amt ) )
      {
        acct.deposit ( amt ) ;
        _RvLogger._rvLogEnd(this, "transfer", "!double^0!Account^0!") ;
        return ( true ) ;
      }
      _RvLogger._rvLogEnd(this, "transfer", "!double^0!Account^0!") ;
      return ( false ) ;
    }
  }

  // REVY added 1 field and 5 methods
  private int _rvOID = 0 ;
  public String _rvClassName() { return("Account") ; }
  public int _rvGetOID() { return(_rvOID) ; }
  public void _rvSetOID(int _i) { _rvOID = _i ; }
  public int _rvStaticSize() { return(16) ; }
  public int _rvRuntimeSize(Hashtable _vset)
  {
    int _acc = 0 ;
    if(_vset.contains(this)) { return(4) ; }
    _vset.put(this, this) ;
    _acc += 4 ; // ACCOUNTNUMBER
    _acc += 4 ; // accountNumber
    _acc += 8 ; // balance
    return(_acc) ;
  }
}
```

APPENDIX C: User's Code with Fast Instrumentation

```
/* ********************** Main.java ********************** */

// REVY imported classes
import java.util.Hashtable ;
import whui.revy.util._RvObject ;
import whui.revy.util._RvLogger ;


public class Main implements _RvObject
{
   public static void main ( String argv [ ] )
   {
     _RvLogger._rvLogInit("Main") ;
     {
       Person will , duane , jonathan ;
       Account willAcct , duaneAcct , jonathanAcct ;
       will = new Person ( "Will" ) ;
       willAcct = new Account ( 1000.0 ) ;
       will.setAccount ( willAcct ) ;
       duane = new Person ( "Duane" ) ;
       duaneAcct = new Account ( 1000.0 ) ;
       duane.setAccount ( duaneAcct ) ;
       jonathan = new Person ( "Jonathan" ) ;
       jonathanAcct = new Account ( 1000.0 ) ;
       jonathan.setAccount ( jonathanAcct ) ;
       duaneAcct.transfer ( 100.0 , willAcct ) ;
       jonathanAcct.transfer ( 100.0 , willAcct ) ;
       System.out.println ( will ) ;
       System.out.println ( duane ) ;
       System.out.println ( jonathan ) ;
     }
     _RvLogger._rvLogDone("Main") ;
   }

   // REVY added 1 field and 5 methods
   private int _rvOID = 0 ;
   public String _rvClassName() { return("Main") ; }
   public int _rvGetOID() { return(_rvOID) ; }
   public void _rvSetOID(int _i) { _rvOID = _i ; }
   public int _rvStaticSize() { return(0) ; }
   public int _rvRuntimeSize(Hashtable _vset)
   {
     int _acc = 0 ;
     if(_vset.contains(this)) { return(4) ; }
     _vset.put(this, this) ;
     return(_acc) ;
   }
}


/* ********************** Person.java ********************** */

// REVY imported classes
import java.util.Hashtable ;
import whui.revy.util._RvObject ;
import whui.revy.util._RvLogger ;


public class Person extends Object implements _RvObject
{
   public static int SIN = 123456789 ;
   public int sin ;
   public String name ;
```

```
    public Account account ;
    public Person ( String n )
    {
      this.sin = SIN ++ ;
      this.name = n ;
      _RvLogger._rvLogCreate(this) ;
    }
    public String toString ( )
    {
      return ( "<Person sin=" + this.sin + " name=" + this.name + ">/n" + account.toString ( ) ) ;
    }
    public int getSIN ( ) { return ( this.sin ) ; }

    public String getName ( ) { return ( this.name ) ; }

    public Account getAccount ( ) { return ( this.account ) ; }

    public void setAccount ( Account acct ) { this.account = acct ; }

    // REVY added 1 field and 5 methods
    private int _rvOID = 0 ;
    public String _rvClassName() { return("Person") ; }
    public int _rvGetOID() { return(_rvOID) ; }
    public void _rvSetOID(int _i) { _rvOID = _i ; }
    public int _rvStaticSize() { return(56) ; }
    public int _rvRuntimeSize(Hashtable _vset)
    {
      int _acc = 0 ;
      if(_vset.contains(this)) { return(4) ; }
      _vset.put(this, this) ;
      _acc += 4 ; // SIN
      _acc += 4 ; // sin
      _acc += 32 ; // name
      _acc += ((account == null)?  _NULL_SZE:account._rvRuntimeSize(_vset)) ; // account
      return(_acc) ;
    }
}


/* ********************** Account.java ********************** */

// REVY imported classes
import java.util.Hashtable ;
import whui.revy.util._RvObject ;
import whui.revy.util._RvLogger ;


public class Account extends Object implements _RvObject
{
   public static int ACCOUNTNUMBER = 654321 ;
   public double balance ;
   public Account ( double amt )
   {
     this.accountNumber = ACCOUNTNUMBER ++ ;
     this.balance = amt ;
     _RvLogger._rvLogCreate(this) ;
   }
   public String toString ( )
   {
     return ( "<Account AccoutNumber=" + this.accountNumber + " balance=" + this.balance + ">" ) ;
   }
   public int getAccountNumber ( ) { return ( this.accountNumber ) ; }

   public double getBalance ( ) { return ( this.balance ) ; }

   public void deposit ( double amt )
   {
     int _sze = 0 ;
```

```
      _size += 8 ;
      _RvLogger._rvLogStart(this, "deposit", "!double^0!", _sze) ;
      {
        this.balance += amt ;
      }
      _RvLogger._rvLogEnd(this, "deposit", "!double^0!") ;
    }
    public boolean withdraw ( double amt )
    {
      int _sze = 0 ;
      _size += 8 ;
      _RvLogger._rvLogStart(this, "withdraw", "!double^0!", _sze) ;
      {
        if ( this.balance < amt )
        {
          System.err.println ( "Sorry, not enough money for withdrawal" ) ;
          _RvLogger._rvLogEnd(this, "withdraw", "!double^0!") ;
          return ( false ) ;
        }
        this.balance -= amt ;
        _RvLogger._rvLogEnd(this, "withdraw", "!double^0!") ;
        return ( true ) ;
      }
    }
    public boolean transfer ( double amt , Account acct )
    {

      int _sze = 0 ;
      _size += 8 ;
      _size += ((acct == null)?  _NULL_SZE:acct._rvStaticSize()) ;
      _RvLogger._rvLogStart(this, "transfer", "!double^0!Account^0!", _sze) ;
      {
        if ( this.withdraw ( amt ) )
        {
          acct.deposit ( amt ) ;
          _RvLogger._rvLogEnd(this, "transfer", "!double^0!Account^0!") ;
          return ( true ) ;
        }
        _RvLogger._rvLogEnd(this, "transfer", "!double^0!Account^0!") ;
        return ( false ) ;
      }
    }

    // REVY added 1 field and 5 methods
    private int _rvOID = 0 ;
    public String _rvClassName() { return("Account") ; }
    public int _rvGetOID() { return(_rvOID) ; }
    public void _rvSetOID(int _i) { _rvOID = _i ; }
    public int _rvStaticSize() { return(16) ; }
    public int _rvRuntimeSize(Hashtable _vset)
    {
      int _acc = 0 ;
      if(_vset.contains(this)) { return(4) ; }
      _vset.put(this, this) ;
      _acc += 4 ; // ACCOUNTNUMBER
      _acc += 4 ; // accountNumber
      _acc += 8 ; // balance
      return(_acc) ;
    }
}
```

APPENDIX D: Revy Project File (RVY)

```
/usr/maligne2/grad/whui/java/classes/whui/revy/eg/account/Main.java
/usr/maligne2/grad/whui/java/classes/whui/revy/eg/account/Person.java
/usr/maligne2/grad/whui/java/classes/whui/revy/eg/account/Account.java
project.compile=javac /usr/maligne2/grad/whui/java/classes/whui/revy/eg/account/Main.java
project.execute=java -O -classpath /usr/maligne2/grad/whui/java/classes/whui/revy/eg/account Main
```

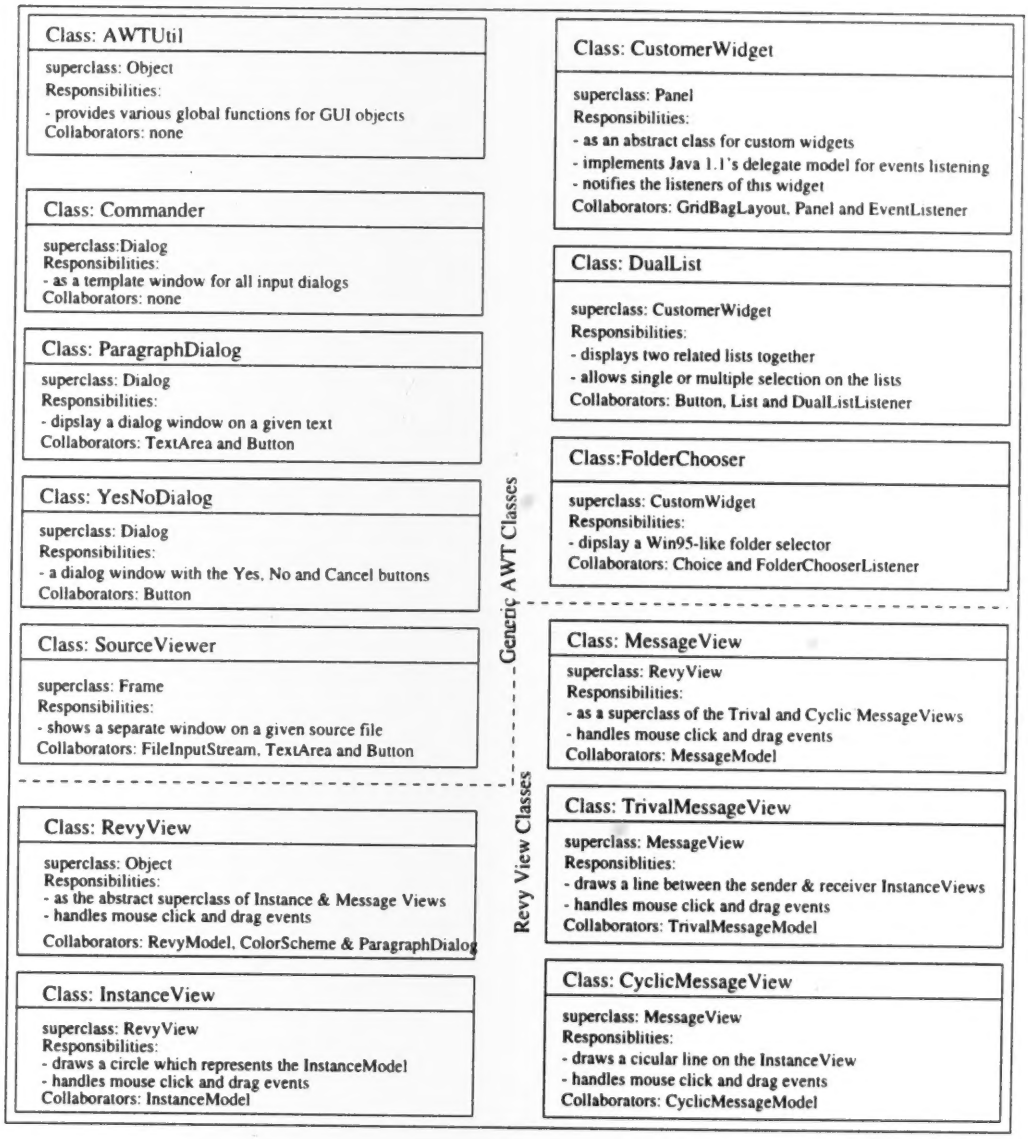
APPENDIX C: Static Artifacts File (SAF)

```
<Project name=Main>

   <Source name=/usr/maligne2/grad/whui/java/classes/whui/revy/eg/account/Main.java>
   <Source name=/usr/maligne2/grad/whui/java/classes/whui/revy/eg/account/Person.java>
   <Source name=/usr/maligne2/grad/whui/java/classes/whui/revy/eg/account/Account.java>

   <Type name=Account classification=2 byteSize=16>
   <Type name=int classification=0 byteSize=4>
   <Type name=byte classification=0 byteSize=1>
   <Type name=float classification=0 byteSize=4>
   <Type name=Main classification=2 byteSize=0>
   <Type name=double classification=0 byteSize=8>
   <Type name=long classification=0 byteSize=8>
   <Type name=boolean classification=0 byteSize=0>
   <Type name=char classification=0 byteSize=2>
   <Type name=short classification=0 byteSize=2>
   <Type name=void classification=0 byteSize=0>
   <Type name=Person classification=2 byteSize=56>

   <Class name=Account superClasses=!Object!  annotated=yes>
     <Field name=ACCOUNTNUMBER class=Account type=int dimension=0>
     <Field name=accountNumber class=Account type=int dimension=0>
     <Field name=balance class=Account type=double dimension=0>
     <Method name=toString class=Account type=_system_ dimension=0 annotated=no>
     </Method>
     <Method name=getAccountNumber class=Account type=int dimension=0 annotated=no>
     </Method>
     <Method name=getBalance class=Account type=double dimension=0 annotated=no>
     </Method>
     <Method name=deposit class=Account type=void dimension=0 annotated=yes>
       <Parameter name=amt method=deposit type=double dimension=0>
     </Method>
     <Method name=withdraw class=Account type=boolean dimension=0 annotated=yes>
       <Parameter name=amt method=withdraw type=double dimension=0>
     </Method>
     <Method name=transfer class=Account type=boolean dimension=0 annotated=yes>
       <Parameter name=amt method=transfer type=double dimension=0>
       <Parameter name=acct method=transfer type=Account dimension=0>
     </Method>
   </Class>

   <Class name=Main superClasses=!  annotated=no>
   </Class>

   <Class name=Person superClasses=!Object!  annotated=yes>
     <Field name=SIN class=Person type=int dimension=0>
     <Field name=sin class=Person type=int dimension=0>
     <Field name=name class=Person type=_system_ dimension=0>
     <Field name=account class=Person type=Account dimension=0>
     <Method name=toString class=Person type=_system_ dimension=0 annotated=no>
     </Method>
     <Method name=getSIN class=Person type=int dimension=0 annotated=no>
     </Method>
     <Method name=getName class=Person type=_system_ dimension=0 annotated=no>
     </Method>
     <Method name=getAccount class=Person type=Account dimension=0 annotated=no>
     </Method>
     <Method name=setAccount class=Person type=void dimension=0 annotated=no>
       <Parameter name=acct method=setAccount type=Account dimension=0>
     </Method>
   </Class>

</Project>
```

APPENDIX F: Dynamic Trace File (DTF)

```
##### Revy-Logger:  starts @ Wed Nov 19 12:03:58 PDT 1997
<Crt cls=Person oid=1 tid=0 t=289>
<Crt cls=Account oid=2 tid=0 t=295>
<Crt cls=Person oid=3 tid=0 t=296>
<Crt cls=Account oid=4 tid=0 t=297>
<Crt cls=Person oid=5 tid=0 t=298>
<Crt cls=Account oid=6 tid=0 t=299>
<Msg cls=Account oid=4 mth=transfer sgn=!double^0!Account^0!  sze=24 tid=0 t=301>
<Msg cls=Account oid=4 mth=withdraw sgn=!double^0!  sze=8 tid=0 t=302>
</Msg cls=Account oid=4 mth=withdraw sgn=!double^0!  tid=0 t=303>
<Msg cls=Account oid=2 mth=deposit sgn=!double^0!  sze=8 tid=0 t=304>
</Msg cls=Account oid=2 mth=deposit sgn=!double^0!  tid=0 t=305>
</Msg cls=Account oid=4 mth=transfer sgn=!double^0!Account^0!  tid=0 t=306>
<Msg cls=Account oid=6 mth=transfer sgn=!double^0!Account^0!  sze=24 tid=0 t=308>
<Msg cls=Account oid=6 mth=withdraw sgn=!double^0!  sze=8 tid=0 t=309>
</Msg cls=Account oid=6 mth=withdraw sgn=!double^0!  tid=0 t=310>
<Msg cls=Account oid=2 mth=deposit sgn=!double^0!  sze=8 tid=0 t=311>
</Msg cls=Account oid=2 mth=deposit sgn=!double^0!  tid=0 t=312>
</Msg cls=Account oid=6 mth=transfer sgn=!double^0!Account^0!  tid=0 t=313>
##### Revy-Logger:  done @ Wed Nov 19 12:03:58 PDT 1997
```

APPENDIX G: CRC Cards for VIS Supporting Classes



APPENDIX H: CRC Cards for VIS Major Classes

Class: RevyWindow

superclass: Frame
Responsibilities:
- main GUI between the user and the Revy system
- communicates between the user, PAS and RMS
- pops up dialog, draws buttons, handles menu items, etc

Collaborators: UserProject, ObjectGraph, InteractionDisplay, Annotator, DTFReader and the 5 Commanders

Class: InteractionDisplay

superclass: Canvas
Responsibilities:
- draws the instance and message views
- receives windows events and relays to corresponding views
- calls for animation and adjust the animation settings

Collaborators: ObjectGraph, RevyModel, RevyView and MessageAnimator

Class: MessageAnimator

superclass: Thread
Responsibilities:
- draws the animation on a MessageView object
- allows the setting of animation speed

Collaborators: MessageView

Class: ColorScheme

superclass: Object
Responsibilities:
- defines the color based on the activity %

Collaborators: RevyModel and Color

Class: ProjectCommander

superclass: Commander
Responsibilities:
- allows users to define the project parameters

Collaborators: FolderChooser and UserProject

Class: AnnotationCommander

superclass: Commander
Responsibilities:
- allows users to select the classes & methods to instrument

Collaborators: DualList and UserProject

Class: ShellCommander

superclass: Commander
Responsibilities:
- allows users to edit the compile and execute commands

Collaborators: TextArea and UserProject

Class: AnimationCommander

superclass: Commander
Responsibilities:
- allows users to adjust the animation settings

Collaborators: Scrollbar and InteractionDisplay

Class: FilteringCommander

superclass: Commander
Responsibilities:
- allows users to select the instances & methods to filter out

Collaborators: DualList, UserProject and ObjectGraph

APPENDIX I: CRC Cards for PAS Supporting Classes

Class: UserCode

superclass: Object
Responsibilities:
- acts as an abstract class for its subclasses
- all its subclasses store the static artifacts of the user's code

Collaborators: none

Class: UserClass

superclass: UserCode, implements: UserAnnotatable
Responsibilities:
- stores user's superclasses, methods and fields
- provides annotation interfaces on methods
- provides annotation routines on the class
- computes the static size (i.e. lower bound size)
- generates code for computing runtime object size

Collaborators: UserType, UserMethod and UserField

Class: UserFile

superclass: UserCode
Responsibilities:
- superclass for objects that directly manipulates files
- provides standard file manipulation routines

Collaborators: File, IOStream

Class: UserEvaluable

superclass: UserCode
Responsibilites:
- as an abstract superclass for any evaluable object, e.g. method, parameter
- stores the static type and the declaring module of the evaluable object

Collaborators: UserType & UserClass

Class: UserProject

superclass: UserFile
Responsibilities:
- manipulates the RVY project file
- generates and reads SAF file
- stores all user source files, types and classes
- provides annotation interfaces on classes and methods

Collaborators: UserSource, UserType & UserClass

Class: UserMember

superclass: UserEvaluable
Responsibilities:
- a UserMember is a member object of a class; it can be either a field or method

Collaborators: UserType & UserClass

Class: UserSource

superclass: UserFile
Responsibilities:
- stores the user source file names
- backups user original sources

Collaborators: System & Runtime

Class: UserField

superclass: UserMember
Responsibilities:
- stores the details of a field of a class
- generates field size code

Collaborators: UserType & UserClass

Class: UserType

superclass: UserCode
Responsibilities:
- predefines the primitive types
- stores the user's type name and size

Collaborators: none

Class: UserMethod

superclass: UserMember, implements: UserAnnotatable
Responsibilities:
- stores the details, like parameters, of a method
- provides annotation routines on the method
- generates method signature
- generates code for computing size of parameters

Collaborators: UserType, UserClass & UserParameter

Interface: UserAnnotatable

superclass: none
Responsibilities:
- defines the annotation interface routines

Collaborators: none

Class: UserParameter

superclass: UserEvaluable
Responsibilities:
- stores the details of a parameter
- generates parameter size code

Collaborators: UserType

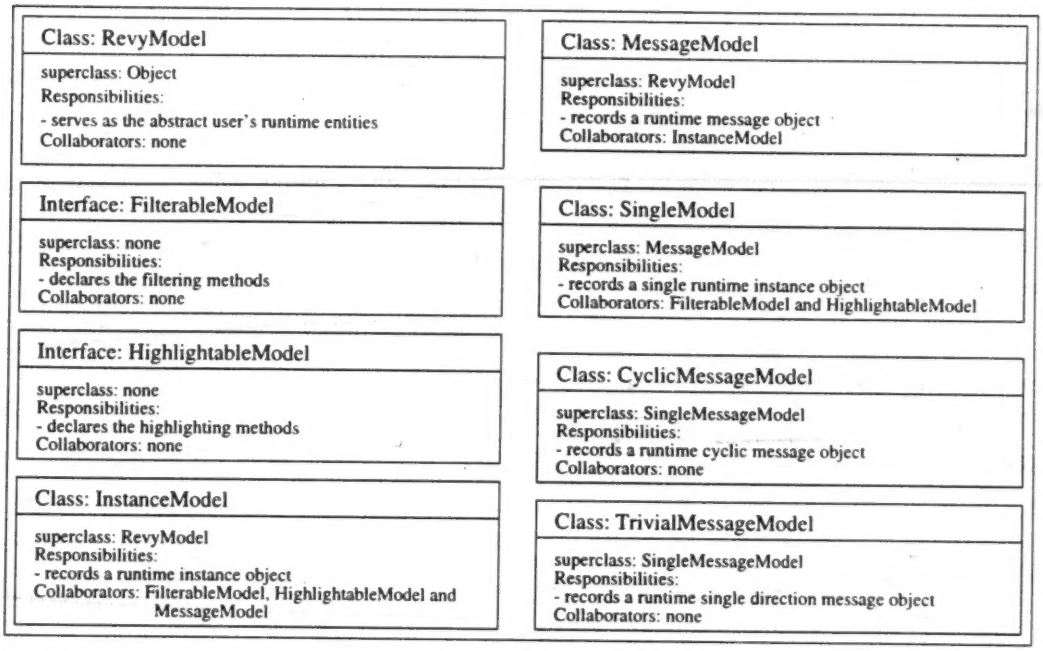
APPENDIX J: CRC Cards for PAS Major Classes

Class: Annotator
superclass: Object Responsibilities: - receives requests from the main Revy system - distributes services to either JavaSourcer or C++Sourcer Collaborators: JavaSourcer, C++Sourcer, and UserProject

Class: C++Sourcer
superclass: none Responsibilities: - same as JavaSourcer execpt it works for C++ sources

Class: JavaSourcer
superclass: Object Responsinbilities: - reads the RVY project file - parses the user's Java source files - creates SAF file based on the type, class, fields and methods declared in the user's sources - instruments the code according to the directions given in the user modified SAF file - can be invoked with two options: 1) -getSAF or -annotate: parsing or instrumenting 2) -annotateBetter or -annotateFaster: accurate or faster Collaborations: all User* objects

APPENDIX K: CRC Cards for RMS Supporting Classes



Class: RevyModel

superclass: Object
Responsibilities:
- serves as the abstract user's runtime entities
Collaborators: none

Interface: FilterableModel

superclass: none
Responsibilities:
- declares the filtering methods
Collaborators: none

Interface: HighlightableModel

superclass: none
Responsibilities:
- declares the highlighting methods
Collaborators: none

Class: InstanceModel

superclass: RevyModel
Responsibilities:
- records a runtime instance object
Collaborators: FilterableModel, HighlightableModel and MessageModel

Class: MessageModel

superclass: RevyModel
Responsibilities:
- records a runtime message object
Collaborators: InstanceModel

Class: SingleModel

superclass: MessageModel
Responsibilities:
- records a single runtime instance object
Collaborators: FilterableModel and HighlightableModel

Class: CyclicMessageModel

superclass: SingleMessageModel
Responsibilities:
- records a runtime cyclic message object
Collaborators: none

Class: TrivialMessageModel

superclass: SingleMessageModel
Responsibilities:
- records a runtime single direction message object
Collaborators: none

APPENDIX L: CRC Cards for RMS Major Classes

Interface: _RvObject
superclass: Object Responsibilities: - acts as a superclass for user's annotated classes - declares the interface methods Collaborators: none

Class: ObjectGraph
superclass: Object Responsibilities: - stores the runtime call graph details - reports the runtime statistics Collaborators: InstanceModel and MessageModel

Class: _RvLogger
superclass: Object Responsibilities: - keeps track of the OID counter - assigns OID upon the creation of an object - logs the creation, destruction, method start and end to a DTF file

Class: DTFReader
superclass: Object Responsibilities: - parses a Dynamic Trace File - computes the runtime statistics Collaborators: InstanceModel, MessageModel and ObjectGraph

Bibliography

[AM87] P. America, *POOL-T: A Parallel Object-Oriented Languages*, in A. Yonezewa and M. Tokoroi (editors), *Object-Oriented Concurrent Programming*, pp.199-220, MIT Press, 1987.

[AC96] I. Attali, D. Caromel, S.O. Ehmety and S. Lippi, *Semantic-Based Visualization for Parallel Object-Oriented Programming*, OOPSLA '96, pp. 421-440, 1996 (also at: http://www.inria.fr/croap/eiffel-ll).

[BC88] P. Borras, D. Clment, Th. Despeyroux, J. Incerpi, G. Kahn, B. Lang and V. Pascual, *Centaur: The System*, SIGSOFT '88, 1988 (also at: http://www.inria.fr/croap/centaur/centaur.html).

[BE97] D. Bell, *Make Java fast: Optimize!*, JavaWorld, April 1997 (http://www.javaworld.com/javaworld/jw-04-1997/jw-04-optimize.html).

[BG97] A. J. C. Bik and D. B. Gannon, *Automatically Exploiting Implicit Parallelism in Java*, Concurrency, Practice and Experience, 9(6):579-619, 1997 (also at: http://www.extreme.indiana.edu/hpjava/JAVAR/index.html).

[CA88] Denis Caromel, *A General Model for Concurrent and Distributed Object-Oriented Programming*, Workshop on Object-Based Concurrent Programming, OOPSLA '88, 1988.

[CT92] K. M. Chandy and S. Taylor, *An Introduction To Parallel Programming*, Jones and Bartlett, 1992.

[ER95] Extreme Research Group of Indiana University, *Sage++ WWW Page*, http://www.extreme.indiana.edu/sage/index.html, 1995.

[FL96] D. Flanagan, *Java in a Nutshell*, O'Reilly Publishing, 1996.

[GH94] E. Gamma, R. Helm, R. Johnson and J. Vlissides, *Design Patterns - Elements of Reusable Object-Oriented Software*, Addison-Wesley Press, 1994.

[GH97] M. Green and J. Hoover, *U of Alberta CMPUT 301 Course Notes - Approaches to User Interface Design*, http://www.cs.ualberta.ca/~mark/c301/UIDesign.html, 1997.

[GW95] G. V. Wilson, *Practical Parallel Programming*, The MIT Press, 1995.

[GJ96] J. Gosling, B. Joy and G. Steele, *The Java Language Specification*, Addison-Wesley Press, 1996.

[HA97] J. Hardwick, *Java Microbenchmarks*, http://www.cs.cmu.edu/~jch/java/benchmarks.html, 1997.

[HE91] M. T. Heath and J. A. Etherisge, *Visualizing Performance of Parallel Programs*, IEEE Software 8-5:29-29, 1991.

[HU95] P. S. Huang, *Selecting Better-Performing Alternative Code Using Run-time Profiling Feedback*, Master's Thesis, Massachusetts Institute of Technology, June 1996.

[IM95] P. Iglinski, S. MacDonald, C. Morrow, D. Novillo, I. Parsons, J.Schaeffer, D. Szafron and D. Wolosschuk, *Enterprise User's Manual Version 2.4. Technical Report TR 95-02*, University of Alberta, January 1995.

[JS97] JavaSoft Corporation, *Java Development Kit WWW Page*, http://www.javasoft.com/products/jdk/1.1/, 1997.

[KL89] D. Kafura and K.H. Lee, *Inheritance in Actor Based Concurrent Object-Oriented Languages*, ECOOP '89, pp. 131-145, Cambridge University Press, 1989.

[KO96] D. Kondratiev, *HTML Parser for JDK 1.0*, http://aldan.paragraph.com/JavaApp/JavaApp.htm, 1996.

[KP89] G. E. Krasner and S. Pope, *A Cookbook for Using the Model-View-Controller User Interface Paradigm in Smalltalk-80*, Jornal of Object-oriented Programming, 1(3):26, 1989.

[KS93] E. Kraemer and J.T. Stasko, *The Visualization of Parallel Systems: An Overview*, Journal of Parallel and Distributed Computing 18:105-117, 1993.

[LA96] C. Laffra, *Advanced Java - Idioms, Pitfalls, Styles, and Programming Tips*, Prentice Hall, 1996.

[LE96] J. Leach, *JHLZip - Another .zippy Utility*, http://www.easynet.it/~jhl/apps/zip/zip.html, 1996.

[LE97] D. Lea, *Concurrent Programming in Java*, Addison-Wesley, 1997.

[LR95] J. Lamping, R. Rao and P. Pirolli, *A Focus+Context Technique Based on Hyperbolic Geometry for Visualizing Large Hierarchies*, CHI '95 Conference Proceedings, 1995 (also at: http://www.acm.org/sigchi/chi95/Electronic/documnts/papers/jl_bdy.htm).

[MA96] S. MacDonald, *An OO Runtime System for Parallel Programming*, Master's Thesis, University of Alberta, 1996.

[MA97] S. MacDonald, *The MethodThread Class for Java*, http://www.cs.ualberta.ca/~stevem/MethodThread/index.html, 1997.

[MB95] T. Munzner and P. Burchard, *Visualizing the Structure of the World Wide Web in 3D Hyperbolic Space*, SIGGRAPH VRML '95, pp. 33-38, 1995.

[MY94] S. Matsuoka and A. Yonezewa, *Analysis of Inheritance Anomaly in Object-Oriented Concurrent Programming*, in G. Agha, P. Wegner and A. Yonezewa (editors), *Research Directions in Concurrent Object-Oriented Programming*, pp. 107-150, MIT Press, 1993.

[OS94] ObjectSoftware Inc., *ObjectTrace - Release 2.0 - A Tool to Generate C++ Application Trace*, http://www.obsoft.com/PostScript/ObjectTraceTwo.ps.Z.

[SB96] A.D. Stoyenko, J. Bosch, M. Aksit and T.J. Marlow, *Load Balanced Mapping of Distributed Objects to Minimize Network Communication*, Journal of Parallel and Distributed Computing 34:117-136, 1996.

[SS93] J. Schaeffer, D. Szafron, G. Lobe and I. Parsons, *The Enterprise model for Developing Distributed Applications*, IEEE Parallel and Distributed Technology, 1(3):85-96, 1993.

[SS97] SunSoft Corporation, *Java On Solaris 2.6 – A White Paper*, http://www.sun.com/solaris/java/wp-java/, 1997.

[ST97] SunTest – The Java Testing Unit of Sun Microsystems, *Java Compiler Compiler WWW Page*, http://www.suntest.com/JavaCC/index.html, 1997.

[SU97] The Stanford SUIF Compiler Group, *SUIF WWW Page*, http://suif.stanford.edu/, 1997.

[TU94] G. T. and C. W. Ueberhuber, *Visualization of Scientific Parallel Programs*, Springer-Verlag, 1994.

[WH96] G. White, *ProfileViewer*, http://www.inetmi.com/~gwhi/ProfileViewer/ProfileViewer.html.

[WI95] D. Woloschuk, P. Iglinski, S. MacDonald, D. Novillo, I. Parsons, J. Schaeffer and D. Szafron, *Performance Debugging in the Enterprise Parallel Programming System*, CASCON '95, 1995.

[WW94] J. Waldo, G. Wyant, A. Wollrath and S. Kendall, *A Note on Distributed Computing*, Technical Report SMLI TR-94-29, Sun Microsystems, November, 1994.

[YO96] D. Young, *Java Performance*, Silicon Graphics European Developer Forum '96, (also at: http://www.sgi.com/Support/DevProg/Forum/eurof96/presents/webucator/Young3/), 1996.

[YT86] Y. Yokote and M. Tokoro, *The Design and Implementation of Concurrent Smalltalk*, OOPSLA '86, pp.331-340, 1986.